ADL User Guide for Open AT® OS v6.00

Revision: 003
Date: December 2007

OPEN AT

wavecom
Make it wireless

Operating Systems | Integrated Development Environments | Plug-Ins | Wireless CPUs | Services

ADL User Guide for Open AT® OS v6.00

Reference: WM_DEV_OAT_UGD_060
Revision: 003
Date: December 17, 2007

Document History

Index	Date	Versions	
001	October 3, 2007	Creation	
002	November 20, 2007	Update	
003	December 17, 2007	Update	

Copyright

Trademarks

, , ®, “YOU MAKE IT, WE MAKE IT WIRELESS®”, WAVECOM®, Wireless Microprocessor®, Wireless CPU®, Open AT® and certain other trademarks and logos appearing on this document, are filed or registered trademarks of Wavecom S.A. in France and/or in other countries. All other company and/or product names mentioned may be filed or registered trademarks of their respective owners.

No Warranty/No Liability

This document is provided “as is”. Wavecom makes no warranties of any kind, either expressed or implied, including any implied warranties of merchantability, fitness for a particular purpose, or noninfringement. The recipient of the documentation shall endorse all risks arising from its use. In no event shall Wavecom be liable for any incidental, direct, indirect, consequential, or punitive damages arising from the use or inadequacy of the documentation, even if Wavecom has been advised of the possibility of such damages and to the extent permitted by law.

Overview

This user guide describes the Application Development Layer (ADL). The aim of the Application Development Layer is to ease the development of Open AT® embedded application. It applies to revision Open AT® 6.00 and higher (until next version of this document).

Table of Contents

Document History 2
Copyright 3
Trademarks 3
No Warranty/No Liability 3
Overview 4
Table of Contents 5
List of Figures 14
1 Introduction 15
1.1 Important Remarks 15
1.2 References 15
1.3 Glossary 15
1.4 Abbreviations 16
2 Description 17
2.1 Software Architecture 17
2.2 ADL Limitations 18
2.3 Open AT® Memory Resources 18
2.3.1 RAM Resources 18
2.3.2 Flash Resources 18
2.4 Defined Compilation Flags 19
2.5 Inner AT Commands Configuration 19
2.6 Open AT® Specific AT Commands 20
2.6.1 AT+WDWL Command 20
2.6.2 AT+WOPEN Command 20
2.7 Notes on Wavecom Firmware 21
2.8 Security 21
2.8.1 Software Security: Memory Access Protection 21
2.8.2 Hardware Security: Watchdog Protection 22
2.8.3 Safe Boot Mode 22
2.9 RTE limitations 22
2.9.1 Sending large buffers through an ADL API 22
2.9.2 IRQ Services 22
3 API 23
3.1 Application Entry Points Interface 23
3.1.1 Required Header File 23
3.1.2 The adl_InitTasks_t Structure 23
3.1.3 Tasks Definition Table 25
3.1.4 Interrupt Handlers Call Stack Sizes Declaration 26

3.1.5 The adl_InitType_e Type 27
3.1.6 The adl_InitGetType function 28
3.1.7 Example 28
3.2 Basic Features 29
3.2.1 Data Types 29
3.2.2 List Management 29
3.2.3 Standard Library 35
3.2.4 Sound API 38
3.3 AT Commands Service 46
3.3.1 Required Header File 46
3.3.2 Unsolicited Responses 46
3.3.3 Responses 49
3.3.4 Incoming AT Commands 51
3.3.5 Outgoing AT Commands 61
3.4 Timers 68
3.4.1 Required Header Files 68
3.4.2 The adl_tmr_t Structure 68
3.4.3 Defines 69
3.4.4 The adl_tmrType_e 69
3.4.5 The adl_tmrCyclicMode_e 70
3.4.6 The adl_tmr_Handler_t 71
3.4.7 The adl_tmrSubscribe Function 71
3.4.8 The adl_tmrSubscribeExt Function 72
3.4.9 The adl_tmrUnSubscribe Function 73
3.4.10 Example 75
3.5 Memory Service 76
3.5.1 Required Header File 76
3.5.2 Data Structures 76
3.5.3 Defines 78
3.5.4 The adl_memGetInfo Function 78
3.5.5 The adl_memGet Function 79
3.5.6 The adl_memRelease Function 80
3.5.7 Heap Memory Block Status 80
3.5.8 Example 80
3.6 ADL Registry Service 81
3.6.1 Required Header File 81
3.6.2 The adl_regGetWCPUType Function 81
3.6.3 The adl_regGetHWInteger Function 81
3.6.4 The adl_regGetHWData Function 82
3.6.5 The adl_regGetHWDataChunk Function 82
3.6.6 Example 83
3.7 Debug Traces 85
3.7.1 Required Header File 85
3.7.2 Build Configuration Macros 85
3.7.3 The adl_trcPrint Function 86
3.7.4 The adl_trcDump Function 87
3.7.5 Example 89
3.8 Flash 90
3.8.1 Required Header File 90

3.8.2 Flash Objects Management 90
3.8.3 The adl_flhSubscribe Function 91
3.8.4 The adl_flhExist Function 92
3.8.5 The adl_flhErase Function 92
3.8.6 The adl_flhWrite Function 93
3.8.7 The adl_flhRead Function 93
3.8.8 The adl_flhGetFreeMem Function 94
3.8.9 The adl_flhGetIDCount Function 94
3.8.10 The adl_flhGetUsedSize Function 95
3.9 FCM Service 96
3.9.1 Required Header File 97
3.9.2 The adl_fcmIsAvailable Function 97
3.9.3 The adl_fcmSubscribe Function 98
3.9.4 The adl_fcmUnsubscribe Function 101
3.9.5 The adl_fcmReleaseCredits Function 102
3.9.6 The adl_fcmSwitchV24State Function 102
3.9.7 The adl_fcmSendData Function 103
3.9.8 The adl_fcmSendDataExt Function 104
3.9.9 The adl_fcmGetStatus Function 106
3.10 GPIO Service 107
3.10.1 Required Header File 107
3.10.2 GPIO Types 108
3.10.3 The adl_ioGetCapabilitiesList Function 112
3.10.4 The adl_ioEventSubscribe Function 113
3.10.5 The adl_ioHdlr_f Call-back Type 114
3.10.6 The adl_ioEventUnsubscribe Function 115
3.10.7 The adl_ioSubscribe Function 117
3.10.8 The adl_ioUnsubscribe Function 118
3.10.9 The adl_ioSetDirection Function 119
3.10.10 The adl_ioRead Function 119
3.10.11 The adl_ioReadSingle Function 120
3.10.12 The adl_ioWrite Function 121
3.10.13 The adl_ioWriteSingle Function 121
3.10.14 Example 122
3.11 Bus Service 125
3.11.1 Required Header File 125
3.11.2 Capabilities Registry Informations 125
3.11.3 Common Data Structures and Enumerations 128
3.11.4 SPI Bus Subscription Data Structures and Enumerations 131
3.11.5 I2C Bus Subscription Data Structures and Enumerations 136
3.11.6 Parallel Bus Subscription Data Structures and Enumerations 138
3.11.7 IOCtl Operations Data Structures and Enumerations 145
3.11.8 Read/Write Data Structures 151
3.11.9 The adl_busSubscribe Function 151
3.11.10 The adl_busUnsubscribe Function 152
3.11.11 The adl_busIOCtl Function 153
3.11.12 The adl_busRead Function 153
3.11.13 The adl_busReadExt Function 154
3.11.14 The adl_busWrite Function 155
3.11.15 The adl_busWriteExt Function 156
3.11.16 The adl_busDirectRead Function 157

3.11.17 The adl_busDirectWrite Function 158
3.11.18 Example 159
3.12 Error Management 161
3.12.1 Required Header File 161
3.12.2 Enumerations 162
3.12.3 Error event handler 163
3.12.4 The adl_errSubscribe Function 164
3.12.5 The adl_errUnsubscribe Function 164
3.12.6 The adl_errHalt Function 164
3.12.7 The adl_errEraseAllBacktraces Function 165
3.12.8 The adl_errStartBacktraceAnalysis Function 166
3.12.9 The adl_errGetAnalysisState Function 166
3.12.10 The adl_errRetrieveNextBacktrace Function 166
3.12.11 Example 167
3.13 SIM Service 169
3.13.1 Required Header File 169
3.13.2 The adl_simSubscribe Function 169
3.13.3 The adl_simUnsubscribe Function 170
3.13.4 The adl_simGetState Function 171
3.13.5 adl_simEnterPIN Function 171
3.14 Open SIM Access Service 172
3.14.1 Required Header File 172
3.14.2 The adl_osaSubscribe Function 172
3.14.3 The adl_osaHandler_f call-back Type 173
3.14.4 The adl_osaSendResponse Function 175
3.14.5 The adl_osaUnsubscribe Function 176
3.14.6 Example 177
3.15 SMS Service 178
3.15.1 Required Header File 178
3.15.2 The adl_smsSubscribe Function 178
3.15.3 The adl_smsSend Function 180
3.15.4 The adl_smsUnsubscribe Function 181
3.16 Message Service 182
3.16.1 Required Header File 182
3.16.2 The adl_msgIdComparator_e Type 182
3.16.3 The adl_msgFilter_t Structure 183
3.16.4 The adl_msgSubscribe Function 184
3.16.5 The adl_msgHandler_f call-back Type 185
3.16.6 The adl_msgUnsubscribe Function 185
3.16.7 The adl_msgSend Function 186
3.16.8 Example 187
3.17 Call Service 188
3.17.1 Required Header File 188
3.17.2 The adl_callSubscribe Function 188
3.17.3 The adl_callSetup Function 191
3.17.4 The adl_callSetupExt Function 191
3.17.5 The adl_callHangup Function 192
3.17.6 The adl_callHangupExt Function 192
3.17.7 The adl_callAnswer Function 193

3.17.8 The adl_callAnswerExt Function 193
3.17.9 The adl_callUnsubscribe Function 193
3.18 GPRS Service 194
3.18.1 Required Header File 194
3.18.2 The adl_gprsSubscribe Function 194
3.18.3 The adl_gprsSetup Function 197
3.18.4 The adl_gprsSetupExt Function 197
3.18.5 The adl_gprsAct Function 198
3.18.6 The adl_gprsActExt Function 199
3.18.7 The adl_gprsDeact Function 200
3.18.8 The adl_gprsDeactExt Function 200
3.18.9 The adl_gprsGetCidInformations Function 201
3.18.10 The adl_gprsUnsubscribe Function 202
3.18.11 The adl_gprsIsAnIPAddress Function 202
3.18.12 Example 203
3.19 Semaphore ADL Service 205
3.19.1 Required Header File 205
3.19.2 The adl_semGetResourcesCount Function 205
3.19.3 The adl_semSubscribe Function 205
3.19.4 The adl_semConsume Function 206
3.19.5 The adl_semConsumeDelay Function 207
3.19.6 The adl_semProduce Function 207
3.19.7 The adl_semUnsubscribe Function 208
3.19.8 The adl_semIsConsumed Function 208
3.19.9 Example 209
3.20 Application Safe Mode Service 210
3.20.1 Required Header File 210
3.20.2 The adl_safeSubscribe Function 210
3.20.3 The adl_safeUnsubscribe Function 212
3.20.4 The adl_safeRunCommand Function 212
3.21 AT Strings Service 214
3.21.1 Required Header File 214
3.21.2 The adl_strID_e Type 214
3.21.3 The adl_strGetID Function 215
3.21.4 The adl_strGetIDExt Function 215
3.21.5 The adl_strIsTerminalResponse Function 216
3.21.6 The adl_strGetResponse Function 216
3.21.7 The adl_strGetResponseExt Function 217
3.22 Application & Data Storage Service 218
3.22.1 Required Header File 218
3.22.2 The adl_adSubscribe Function 218
3.22.3 The adl_adUnsubscribe Function 219
3.22.4 The adl_adEventSubscribe Function 220
3.22.5 The adl_adEventHdlr_f Call-back Type 220
3.22.6 The adl_adEventUnsubscribe Function 222
3.22.7 The adl_adWrite Function 222
3.22.8 The adl_adInfo Function 223
3.22.9 The adl_adFinalise Function 223
3.22.10 The adl_adDelete Function 224
3.22.11 The adl_adInstall Function 224

3.22.12 The adl_adRecompact Function 225
3.22.13 The adl_adGetState Function 226
3.22.14 The adl_adGetCellList Function 227
3.22.15 The adl_adFormat Function 227
3.22.16 The adl_adFindInit Function 228
3.22.17 The adl_adFindNext Function 229
3.22.18 Example 229

3.23 AT/FCM IO Ports Service 231

3.23.1 Required Header File 231
3.23.2 AT/FCM IO Ports 231
3.23.3 Ports Test Macros 232
3.23.4 The adl_portSubscribe Function 233
3.23.5 The adl_portUnsubscribe Function 234
3.23.6 The adl_portIsAvailable Function 234
3.23.7 The adl_portGetSignalState Function 235
3.23.8 The adl_portStartSignalPolling Function 236
3.23.9 The adl_portStopSignalPolling Function 237

3.24 RTC Service 239

3.24.1 Required Header File 239
3.24.2 RTC service Types 239
3.24.3 Enumerations 241
3.24.4 The adl_rtcGetSecondFracPartStep Function 242
3.24.5 The adl_rtcGetTime Function 242
3.24.6 The adl_rtcSetTime Function 242
3.24.7 The adl_rtcConvertTime Function 242
3.24.8 The adl_rtcDiffTime Function 243
3.24.9 Capabilities 244
3.24.10 Example 244

3.25 IRQ Service 245

3.25.1 Required Header File 245
3.25.2 The adl_irqID_e Type 246
3.25.3 The adl_irqNotificationLevel_e Type 247
3.25.4 The adl_irqPriorityLevel_e Type 247
3.25.5 The adl_irqEventData_t Structure 248
3.25.6 The adl_irqCapabilities_t Structure 249
3.25.7 The adl_irqConfig_t Structure 249
3.25.8 The adl_irqOptions_e type 250
3.25.9 The adl_irqHandler_f Type 251
3.25.10 The adl_irqSubscribe Function 252
3.25.11 The adl_irqSubscribeExt Function 253
3.25.12 The adl_irqUnsubscribe Function 253
3.25.13 The adl_irqSetConfig function 254
3.25.14 The adl_irqGetConfig function 254
3.25.15 The adl_irqGetCapabilities Function 255
3.25.16 Example 256

3.26 TCU Service 258

3.26.1 Required Header File 258
3.26.2 Capabilities Registry Informations 258
3.26.3 Data Structures 259
3.26.4 Enumerators 263

3.26.5 Accurate Timers Service 264
3.26.6 Event Capture Service 266
3.26.7 Event Detection Service 269
3.26.8 The adl_tcuSubscribe Function 271
3.26.9 The adl_tcuUnsubscribe Function 272
3.26.10 The adl_tcuStart Function 273
3.26.11 The adl_tcuStop Function 273

3.27 Extint ADL Service 274

3.27.1 Required Header File 275
3.27.2 The adl_extintConfig_t Structure 275
3.27.3 The adl_extintInfo_t Structure 277
3.27.4 Capabilities 278
3.27.5 The adl_extintSubscribe Function 278
3.27.6 The adl_extintConfig Function 280
3.27.7 The adl_extintGetConfig Function 280
3.27.8 The adl_extintRead function 281
3.27.9 The adl_extintUnsubscribe Function 281
3.27.10 Example 282

3.28 Execution Context Service 284

3.28.1 Required Header File 285
3.28.2 The adl_ctxID_e Type 285
3.28.3 The adl_ctxDiagnostic_e Type 286
3.28.4 The adl_ctxState_e Type 286
3.28.5 The adl_ctxGetID Function 287
3.28.6 The adl_ctxGetTaskID Function 288
3.28.7 The adl_ctxGetTasksCount Function 288
3.28.8 The adl_ctxGetDiagnostic Function 288
3.28.9 The adl_ctxGetState Function 289
3.28.10 The adl_ctxSuspend Function 289
3.28.11 The adl_ctxSuspendExt Function 290
3.28.12 The adl_ctxResume Function 291
3.28.13 The adl_ctxResumeExt Function 291
3.28.14 The adl_ctxSleep Function 292
3.28.15 Example 293

3.29 ADL VariSpeed Service 294

3.29.1 Required Header File 294
3.29.2 The adl_vsMode_e Type 294
3.29.3 The adl_vsSubscribe Function 295
3.29.4 The adl_vsSetClockMode Function 295
3.29.5 The adl_vsUnsubscribe function 296
3.29.6 Example 296

3.30 ADL DAC Service 297

3.30.1 Required Header File 297
3.30.2 Data Structure 297
3.30.3 Defines 298
3.30.4 Enumerations 298
3.30.5 The adl_dacSubscribe Function 298
3.30.6 The adl_dacUnsubscribe Function 299
3.30.7 The adl_dacWrite Function 299
3.30.8 The adl_dacAnalogWrite Function 300

3.30.9 The adl_dacRead Function 300
3.30.10 The adl_dacAnalogRead Function 301
3.30.11 Capabilities 301
3.30.12 Example 302
3.31 ADL ADC Service 304
3.31.1 Required Header File 304
3.31.2 The adl_adcRead Function 304
3.31.3 The adl_adcAnalogRead Function 305
3.31.4 Capabilities 305
3.31.5 Example 306
3.32 ADL Queue Service 307
3.32.1 Required Header File 307
3.32.2 The adl_queueOptions_e Type 307
3.32.3 The adl_queueSubscribe Function 308
3.32.4 The adl_queueUnsubscribe Function 308
3.32.5 The adl_ queueIsEmpty Function 309
3.32.6 The adl_ queuePushItem Function 309
3.32.7 The adl_ queuePopItem Function 310
3.32.8 Example 311
3.33 ADL Audio Service 312
3.33.1 Required Header File 312
3.33.2 Data Structures 312
3.33.3 Defines 315
3.33.4 Enumerations 317
3.33.5 Audio events handler 321
3.33.6 Audio resources control 322
3.33.7 Play a pre-defined audio format 323
3.33.8 Audio stream 336
3.33.9 Stop 342
3.33.10 Set/Get options 344
3.34 ADL Secure Data Storage Service 346
3.34.1 Required Header File 346
3.34.2 Data Structure 347
3.34.3 Defines 348
3.34.4 The adl_sdsWrite Function 348
3.34.5 The adl_sdsRead Function 349
3.34.6 The adl_sdsQuery Function 349
3.34.7 The adl_sdsDelete Function 350
3.34.8 The adl_sdsStats Function 351
3.34.9 The adl_sdsUpdateKeys Function 351
3.34.10 Example 352
3.35 ADL WatchDog Service 353
3.35.1 Required Header File 353
3.35.2 The adl_wdPut2Sleep Function 353
3.35.3 The adl_wdAwake Function 354
3.35.4 Example 354
3.35.5 The adl_wdRearmAppWd Function 355
3.35.6 The adl_wdActiveAppWd Function 355
3.35.7 The adl_wdDeActiveAppWd Function 355
3.35.8 Example 356

3.36 ADL Layer 3 Service 357
3.36.1 Required Header File 357
3.36.2 The adl_L3InfoChannelList_e 357
3.36.3 The Layer3 infoEvent Handler 357
3.36.4 The adl_L3infoSubscribe Function 358
3.36.5 The adl_L3infoUnsubscribe Function 359
3.36.6 Example 359
3.36.7 PLMN SCAN Information Channel Interface 360
4 Error Codes 363
4.1 General Error Codes 363
4.2 Specific FCM Service Error Codes 364
4.3 Specific Flash Service Error Codes 364
4.4 Specific GPRS Service Error Codes 364
4.5 Specific A&D Storage Service Error Codes 364
5 Resources 365

List of Figures

Figure 1: General software architecture 17
Figure 2: Error when trying to send too large a data buffer through an API 22
Figure 4: Open AT® RAM Mapping.. 76
Figure 5: Flow Control Manager Representation 96
Figure 6: Intel Mode Timing - Read Process Example 143
Figure 7: Intel Mode Timing - Write Process Example... 143
Figure 8: Motorola Modes Timing Example 144
Figure 9: A&D cell content install window.. 225
Figure 11: ADL External Interruption Service: Example of Interruption with Debounce Period 274
Figure 12: ADL External Interruption Service: Example of Interruption with Stretching Process 274

1 Introduction

1.1 Important Remarks

- It is strongly recommended before reading this document, to read the ADL User Guide Open AT® 6.00 and specifically the Introduction (chapter 1) for having a better overview of what Open AT® is about.
- The ADL library and the standard embedded Open AT® API layer must not be used in the same application code. As ADL APIs will encapsulate commands and trap responses, applications may enter in error modes if synchronization is no more guaranteed.

1.2 References

[1] AT commands Interface Guide for FW 7.0 (ref WM_DEV_OAT_UGD_059)

[2] Tools Manual for Open AT® IDE 1.04 (ref. WM_DEV_OAT_UGD_045)

1.3 Glossary

Application Mandatory API	Mandatory software interfaces to be used by the Embedded Application.
AT commands	Set of standard modem commands.
AT function	Software that processes the AT commands and AT subscriptions.
Embedded API layer	Software developed by Wavecom, containing the Open AT® APIs (Application Mandatory API, AT Command Embedded API, OS API, Standard API, FCM API, IO API, and BUS API).
Embedded Application	User application sources to be compiled and run on a Wavecom product.
Embedded OS	Software that includes the Embedded Application and the Wavecom library.
Embedded software	User application binary: set of Embedded Application sources + Wavecom library.
External Application	Application external to the Wavecom product that sends AT commands through the serial link.
IDE	Integrated Development Environment
Target	Open AT® compatible product supporting an Embedded Application.
Target Monitoring Tool	Set of utilities used to monitor a Wavecom product.
Receive command pre-parsing	Process for intercepting AT responses.

Send command pre-parsing	Process for intercepting AT commands.
Standard API	Standard set of "C" functions.
Wavecom library	Library delivered by Wavecom to interface Embedded Application sources with Wavecom Firmware functions.
Wavecom Firmware	Set of GSM and open functions supplied to the User.

1.4 Abbreviations

A&D	Application & Data
ADL	Application Development Layer
API	Application Programming Interface
APN	Access Point Name
CID	Context IDentifier
CPU	Central Processing Unit
DAC	Digital Analog Converter
EXTINT	External Interruption
FCM	Flow Control Manager
GPIO	General Purpose Input Output
GGSN	Gateway GPRS Support Node
GPRS	General Packet Radio Service
IP	Internet Protocol
IR	Infrared
KB	Kilobyte
MS	Mobile Station
OS	Operating System
PDP	Packet Data Protocol
PDU	Protocol Data Unit
RAM	Random-Access Memory
ROM	Read-Only Memory
RTK	Real-Time Kernel
SDK	Software Development Kit
SMA	Small Adapter
SMS	Short Message Services

2 Description

2.1 Software Architecture

The Application Development Layer library provides a high level interface for the Open AT® software developer. The ADL set of services has to be used to access all the Wavecom Wireless CPU®s capabilities & interfaces.

The Open AT® environment relies on the following software architecture:

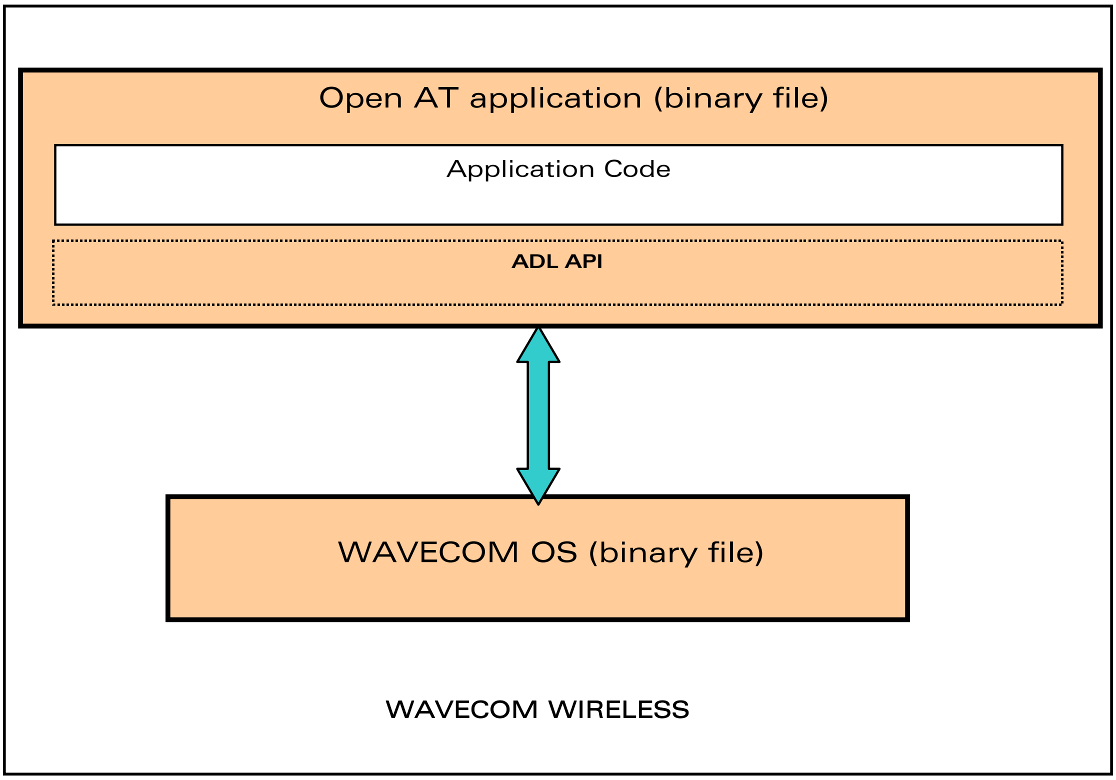


Figure 1: General software architecture

The different software elements on a Wavecom product are described in this section.

The **Open AT® application**, which includes the following items:

- the application code,
- as an option (according to the application needs), one or several Open AT® plug-in libraries (such as the IP connectivity library),
- the Wavecom Application Development Layer library, which provides all the services used by the application,
- the **Wavecom Firmware**, which manages the Wavecom Wireless CPU®.

2.2 ADL Limitations

- ADL is not designed to run in ATQ1 mode (quiet mode, meaning that there is no answer to AT commands).
- While an ADL application is running, the ATQ command always replies +CME ERROR:600 ("Not allowed by embedded application).
- Since ADL uses its own internal process of the +WIND indications, the current value of the AT+WIND command may not be the same when the AT+WOPEN command state is 0 or 1.

2.3 Open AT® Memory Resources

The available memory resources for the Open AT® applications are listed below.

Reminder:

- KB stands for Kilobytes
- MB stands for Megabytes
- Mb stands for Megabits

2.3.1 RAM Resources

The maximum RAM size available for Open AT® applications depends on the Wireless CPU® RAM capabilities, and on the used memory option at project creation time (please refer to the Open AT® IDE Tools Manual for more information [2]):

Total RAM SizeLink Option	8Mb of Total RAM	16Mb of Total RAM or more
"256KB" link option	256KB	256KB
"1MB+" link option	NC*	1MB or more

*"NC" stands for "Not Compatible", i.e. such a linked application will not start if downloaded on such a Wireless CPU®.

2.3.2 Flash Resources

Total Flash Size	ROM(Application code)	Application & Data Storage Volume	Flash Objects Data
32Mb	256-1600KB (default: 832KB)	0-1344KB (default: 768KB)	128KB
64Mb or more	256-(1600+X)KB (default: (832+X)KB)	0-(1344+X)KB (default: 768KB)	128KB

For all flash sizes greater than 32Mb, all additional space is available for A&D and Application Code areas. X stands for this additional flash space in KB. X is reckoned using the following formula:

$X = ((S - 32)/8) * 1024$

Where S is the total Flash size in Mb; E.g. for a 64Mb Flash: X = 4096KB.

The total available flash space for both Open AT® application place and A&D storage place is 1600+X KB.

The maximum A&D storage place size is 1344+X KB (usable for Firmware upgrade capability). In this case the Open AT® application maximum size will be 256 KB.

The minimum A&D storage place size is 0 KB (usable for applications with huge hard coded data).

For more information about the A&D and Application Code areas size configuration, please refer to the AT+WOPEN command description in the AT Commands Interface Guide [1].

Caution:

Any A&D size change will lead to this area format process (some seconds on start-up; all A&D cells data will be erased).

2.4 Defined Compilation Flags

The Open AT® IDE defines some compilation flags, related to the chosen generation environment. Please refer to the Tools Manual for Open AT® IDE [2] for more information.

2.5 Inner AT Commands Configuration

The ADL library needs for its internal processes to set-up some AT command configurations that differ from the default values. The concerned commands are listed hereafter:

AT Command	Fixed value
AT+CMEE	1
AT+WIND	All indications (*)
AT+CREG	2
AT+CGREG	2
AT+CRC	1
AT+CGEREP	2
ATV	1
ATQ	0

(*) All +WIND unsolicited indications are always required by the ADL library. The "+WIND: 3" indication (product reset) will be enabled only if the external application required it.

The above fixed values are set-up internally by ADL. This means that all related error codes (for +CMEE) or unsolicited results are always all available to all Open AT® ADL applications, without requiring them to be sent (using the corresponding configuration command).

Important Caution:

User is strongly advised against modifying the current values of these commands from any Open AT® application. Wavecom would not guarantee ADL correct processing if these values are modified by any embedded application.

External applications may modify these AT commands' parameter values without any constraints. These commands and related unsolicited results behavior is the same with our without a running ADL application.

If errors codes or unsolicited results related to these commands are subscribed and then forwarded by an ADL application to an external one, these results will be displayed for the external application only if this one has required them using the corresponding AT commands (same behavior than the Wavecom AT OS without a running ADL application).

2.6 Open AT® Specific AT Commands

Please refer to the AT Commands Interface Guide (document [1]).

2.6.1 AT+WDWL Command

The AT+WDWL command is usable to download .dwl files trough the serial link, using the 1K Xmodem protocol.

Dwl files may be Wavecom Firmware updates, Open AT® application binaries, or E2P configuration files.

By default this command is not pre-parsed (it can not be filtered by the Open AT® application), except if the Application Safe Mode service is used.

Note:

The AT+WDWL command is described in the document [1].

2.6.2 AT+WOPEN Command

The AT+WOPEN command allows to control Open AT® applications mode & parameters.

Parameters:

0	Stop the application (the application will be stopped on all product resets)
1	Start the application (the application will be started on all product resets)
2	Get the Open AT® libraries versions
3	Erase the objects flash of the Open AT® Embedded Application (allowed only if the application is stopped)
4	Erase the Open AT® Embedded Application (allowed only if the application is stopped)
5	Suspend the Open AT® application, until the AT+WOPENRES command is used, or an hardware interrupt occurs

6 Configures the Application & Data storage place and Open AT® application place sizes.

7 Requires the current Open AT® application state (e.g. to check if the application binary has correctly been built or if the application is running in Target or RTE mode).

8 Configures the Safe Boot mode (refer to §2.8 for more information).

Note:

Refer to the document [1] for more information about this command.

By default this command is not pre-parsed (it can not be filtered by the Open AT® application), except if the Application Safe Mode service is used.

2.7 Notes on Wavecom Firmware

The Open AT® application runs within several tasks managed by the Wavecom Firmware: event handlers are almost always called sequentially by ADL in the first task context, except for the Timers & Messages service (please refer to these services description for more information). The whole ADL API is reentrant and can be called from anymore in the application. If the application offers an API which is supposed to be called from several execution contexts, it is recommended to implement a reentrancy protection mechanism, using the semaphore service

The Wavecom Firmware and the Open AT® application manage their own RAM area. Any access from one of these entities to the other's RAM area is prohibited and causes an exception.

Global variables, call stack and dynamic memory are all part of the RAM allocated to the Open AT® application.

2.8 Security

Security mechanisms are implemented in the Wavecom Firmware in order to protect the Wireless CPU® against software errors. When this occurs, the Wireless CPU® resets and a function call log (called "back-trace") is stored in the Wireless CPU® non-volatile memory. After reset, the **`adl_main`** function is called with the **`ADL_INIT_REBOOT_FROM_EXCEPTION`** value.

After a reset caused by a software crash, the application is started only 20 seconds after the start of the Wavecom Firmware. This allows at least 20 seconds delay to re-download a new application, or to stop the currently running one.

In case of a normal reset, the application restarts immediately.

2.8.1 Software Security: Memory Access Protection

A specific RAM area is allocated to the Open AT® application.

The Open AT® application is seen as a Real-Time task in the Wavecom Firmware, and each time this task runs, the Wavecom RAM protection is activated.

If the Open AT® application tries to access this RAM, then an exception occurs and the software resets.

In case of illegal RAM access, the stored back-trace will display the "**ARM exception 1 xxx**" statement, where "**xxx**" is the address that the application was attempting to access.

2.8.2 Hardware Security: Watchdog Protection

All software (both Open AT® application & Wavecom Firmware) is protected from reaching a dead-end lock by 5 seconds external watchdog reset circuit.

If one task uses the CPU for more than the allowed time, the external watchdog circuit resets the Wireless CPU®.

If a crash due to this watchdog protection is detected, the stored back-trace will display the "**Watchdog Reset**" statement.

2.8.3 Safe Boot Mode

A specific Safe Boot mode is available on the Wireless CPU®.

This mode is activated when a key combination (configured through the AT + WOPEN = 8 command mode) is pressed during Wireless CPU® reset. It is useful when the embedded application causes an exception soon after the Wireless CPU® resets, without any possibility for the external application to send any AT command to disable the Open AT® application.

2.9 RTE limitations

2.9.1 Sending large buffers through an ADL API

Large data buffers (greater than 1600 data bytes) cannot be sent through an ADL API (Eg. adl_busWrite) in RTE mode. If the application tries to do so, an error message (see Figure 2) will be displayed, and the RTE application will stop with an error.

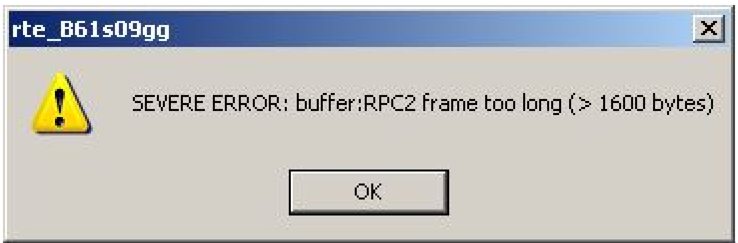


Figure 2: Error when trying to send too large a data buffer through an API

2.9.2 IRQ Services

Due to the RTE architecture and to the very low latency & processing times required in IRQ based applications the IRQ service & all the related services (such as ExtInt services, etc..) are not available in this mode. The subscription function will always fail when called in RTE.

3 API

3.1 Application Entry Points Interface

ADL supplies Application Entry Points Interface to allow applications to define the generic settings of the application tasks and contexts.

The application will have to define its entry points settings using the **adl_InitTasks** table. Each line of this table represents a task, which is characterized by the following parameters:

- the task entry point, called at the Wireless CPU® boot time, in the priority order
- the task call stack size
- the task priority level
- the task name

If the application wishes to use the IRQ service, it will also have to define the call stack sizes for it low level (**adl_InitIRQLowLevelStackSize**) and high level (**adl_InitIRQHighLevelStackSize**) interrupt handlers.

Moreover, some operations related to the initialization are available:

- An Init type check function (**adl_InitGetType**) to retrieve at any time the Wireless CPU® initialization type.

3.1.1 Required Header File

Mandatory application API header file is:

adl_AppliInit.h

(This file is already included by **adl_global.h**)

3.1.2 The adl_InitTasks_t Structure

Open AT® application's tasks declaration structure, used to format the **adl_InitTasks** table.

- Code:

```
typedef struct
{
      void           (* EntryPoint)(void);
      u32            StackSize;
      const ascii*   Name;
      u8             Priority;
} adl_InitTasks_t;
```

- Description

EntryPoint(void)

Task initialization handler, which aims to be called each time the Wireless CPU® boots, as soon as the application is started with the **AT+WOPEN=1** command.

Note:

A task entry point function is NOT like a standard "C" main function. The task does not end when returns. An Open AT® application is stopped only if the **AT+WOPEN=0** command is used. Such a call-back function is only the application entry point, and has to subscribe to some services and events to go further. In addition the whole software is protected by a watchdog mechanism, the application shall not use infinite loops and loops having a too long duration, the Wireless CPU® will reset due to the watchdog hardware security (please refer to Hardware Security: Watchdog Protection for more information).

StackSize

Used to provide to the system the required call stack size (in bytes) for the current task. A call stack is the Open AT® RAM area which contains the local variables and return addresses for function calls. Call stack sizes are deduced from the total available RAM size for the Open AT® application.

Note:

In RTE mode, the call stacks are processed by the host's operating system, and are not configurable (declared sizes are just removed from the available RAM space for the heap memory). It also means that stack overflows cannot be debugged within the RTE mode.

The GCC compiler and GNU Newlib (standard C library) implementation require more stack size than ARM compilers. If the GCC compiler is used, the Open AT® application has to be declared with greater stack sizes.

Call stack sizes shall be declared with some extra bytes margin. It is not recommended to try to reckon exactly the required call stack size of each task.

If the total call stack sizes (including the tasks ones & the interrupt contexts ones) is too large, the Firmware will refuse to launch the application, and the application launch status will be set to 9 (Bad memory configuration)

(cf. **AT+WOPEN=7** description in AT Commands Interface Guide [1] for more information)

Name

Task identification string, used for debug purpose with Traces & Errors services.

Priority

Task priority level, relatively to the other tasks declared in the table. The higher is the number, the higher is the priority level. Priorities values declared in the table should be from 1 to the tasks count. This priority determines the order in which the events are notified to the several tasks when several ones receive information at the same time.

Note:

All the priorities declared in the table have to be different (two tasks can not have the same priority level).

If there is an error in the priorities declaration, the Firmware will refuse to launch the application, and the application launch status will be set to 17 (Bad priority value)

(cf. AT+WOPEN=7 description in AT Commands Interface Guide [1] for more information).

3.1.3 Tasks Definition Table

Mandatory tasks definition table to be provided by the application. For more information on each task's parameters, please refer to the 3.1.2 **adl_InitTasks_t** description. Each line of this table allows to intialize one task. To let the system know how many tasks are required, all the elements of the last line of this table have to be set to 0.

Task entry points declared in the table will be called on Wireless CPU® boot, in the priority order (the highest priority level is called first).

```
Const adl_InitTasks_t  adl_InitTasks[]
```

Note:

At least one task shall be declared in this table. If no tasks are declared in the table, the Firmware will refuse to launch the application, and the application launch status will be set to 16 (No task declared)

(cf. AT+WOPEN=7 description in AT Commands Interface Guide [1] for more information)

There is maximum limit to the number of tasks which shall be declared in this table (Please refer to the Resources chapter for more information. If more tasks than the authorized maximum are declared in the table, the Firmware will refuse to launch the application, and the application launch status will be set to 5 (Too many tasks)

(cf. AT+WOPEN=7 description in AT Commands Interface Guide [1] for more information)

The Multitasking feature has to be enabled on the Wireless CPU® plateform if the application requires more than one task in the table.

If more than one task is declared, and if the feature is not enabled, the Firmware will refuse to launch the application, and the application launch status will be set to 30 (Multitasking feature not enabled)

(cf. AT+WOPEN=7 description in AT Commands Interface Guide [1] for more information)

The Multitasking feature state can be read thanks to the AT+WCFM=5 command response value: Please refer to the AT Commands Inteface guide [1] for more information.

Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU®.

Caution:

Since ADL processing is running in the first application's task context, this one has always to be declared with the highest priority level, otherwise the Firmware will refuse to launch the application, and the application launch status will be set to 11 (Application binary init failure).

(cf. AT+WOPEN=7 description in AT Commands Interface Guide [1] for more information).

3.1.4 Interrupt Handlers Call Stack Sizes Declaration

Interfaces dedicated to the interrupt handlers call stack sizes declaration.

3.1.4.1 Low level interrupt handler call stack size.

Call stack size (in bytes) of the Low level interrupt handler execution context. If the application wishes to handle interruptions (cf. IRQ service chapter & Execution context service chapter), it has also to define the required contexts (low level and/or high level) call stack sizes.

```
const u32 adl_InitIRQLowLevelStackSize
```

Note:

This definition is optional if the application does not plan to use the IRQ service.

The Real Time Enhancement feature has to be enabled on the Wireless CPU® if the application requires this call stack to be greater than zero.

The Real Time Enhancement feature state can be read thanks to the AT+WCFM=5 command response value: Please refer to the AT Commands Interface guide [1] for more information.

Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU®.

If this call stack is declared, and if the feature is not enabled on the Wireless CPU®, the Firmware will refuse to launch the application, and the application launch status will be set to 19 (Real Time feature not enabled)

(cf. AT+WOPEN=7 description in AT Commands Interface Guide [1] for more information).

3.1.4.2 High level interrupt handler call stack size

Call stack size (in bytes) of the High level interrupt handler execution context. If the application whishes to handle interruptions (cf. IRQ service chapter & Execution context service chapter), it has also to define the required contexts (low level and/or high level) call stack sizes.

```
const u32 adl_InitIRQHighLevelStackSize
```

Note:

This definition is optional if the application does not plan to use the IRQ service, or just low level interrupt handlers.

The Real Time Enhancement feature has to be enabled on the Wireless CPU® if the application requires this call stack to be greater than zero.

The Real Time Enhancement feature state can be read thanks to the AT+WCFM=5 command response value: Please refer to the AT Commands Inteface guide [1] for more information.

Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU®.

If this call stack is declared, and if the feature is not enabled on the Wireless CPU®, the Firmware will refuse to launch the application, and the application launch status will be set to 19 (Real Time feature not enabled)

(cf. AT+WOPEN=7 description in AT Commands Interface Guide [1] for more information).

3.1.5 The adl_InitType_e Type

Details of the reason of the Wireless CPU® boot.

- Code

```
typedef enum
{
        ADL_INIT_POWER_ON,
        ADL_INIT_REBOOT_FROM_EXCEPTION,
        ADL_INIT_DOWNLOAD_SUCCESS,
        ADL_INIT_DOWNLOAD_ERROR,
        ADL_INIT_RTC,
} adl_InitType_e;
```

- Description

ADL_INIT_POWER_ON:	Normal power-on.
ADL_INIT_REBOOT_FROM_EXCEPTION:	Reboot after an exception.
ADL_INIT_DOWNLOAD_SUCCESS:	Reboot after a successful install process (cf. **adl_adInstall** API).
ADL_INIT_DOWNLOAD_ERROR:	Reboot after an error in install process (cf. **adl_adInstall** API).
ADL_INIT_RTC:	Power-on due to an RTC alarm (cf. the AT+CALA command documentation for more information).

3.1.6 The adl_InitGetType function

Returns the last Wireless CPU® power-on or reset reason.

- Prototype

  `adl_InitType_e adl_InitGetType (void )`

- Returned values

  The Wireless CPU® reset reason. (please refer to 3.1.5 **adl_InitType_e** description for more information).

- Example:

  This example demonstrates how to use the function **adl_InitGetType** in a nominal case.

```
// Anywhere in the application code, to retrieve init type.
  adl_InitType_e InitType = adl_InitGetType();
```

3.1.7 Example

The code sample below illustrates a nominal use case of the ADL Application Entry Points public interface.

```
// Application tasks declaration table
const adl_InitTasks_t adl_InitTasks [] =
{
    { MyFirstEntryPoint,  1024, "MYTASK1", 3 },
    { MySecondEntryPoint, 1024, "MYTASK2", 2 },
    { MyThirdEntryPoint,  1024, "MYTASK3", 1 },
    { 0, 0, 0, 0 }
};

// Low level handlers execution context call stack size
const u32 adl_InitIRQLowLevelStackSize = 1024;

// High level handlers execution context call stack size
const u32 adl_InitIRQHighLevelStackSize = 1024;
```

3.2 Basic Features

3.2.1 Data Types

The available data types are described in the **wm_types.h** file. They ensure compatibility with the data types used in the functional prototypes and are used for both Target and RTE generation.

3.2.2 List Management

3.2.2.1 Type Definition

3.2.2.1.1 The wm_lst_t Type

This type is used to handle a list created by the list API.

```
typedef void * wm_lst_t;
```

3.2.2.1.2 The wm_lstTable_t Structure

This structure is used to define a comparison callback and an Item destruction callback:

```
typedef struct
{
      s16  ( * CompareItem ) ( void *, void * );
      void ( * FreeItem ) ( void * );
} wm_lstTable_t;
```

The **CompareItem** callback is called every time the list API needs to compare two items.

It returns:

- OK when the two provided elements are considered similar.
- –1 when the first element is considered smaller than the second one.
- 1 when the first element is considered greater than the second one.

If the **CompareItem** callback is set to NULL, the **wm_strcmp** function is used by default.

The **FreeItem** callback is called each time the list API needs to delete an item. It should then perform its specific processing before releasing the provided pointer.

If the **FreeItem** callback is set to NULL, the **wm_osReleaseMemory** function is used by default.

3.2.2.2 The wm_lstCreate Function

The **wm_lstCreate function** allows to create a list, using the provided attributes and callbacks.

- Prototype

```
wm_lst_t wm_lstCreate (u16              Attr,
                       wm_lstTable_t *  funcTable );
```

- Parameters

  Attr:

  List attributes, which can be combined by a logical OR among the following defined values:

  - **WM_LIST_NONE**: no specific attribute ;
  - **WM_LIST_SORTED**: this list is a sorted one (see section 3.2.2.6 **wm_lstAddItem** and section 3.2.2.7 **wm_lstInsertItem** descriptions for more details);
  - **WM_LIST_NODUPLICATES**: this list does not allow duplicate items (see section 3.2.2.6 **wm_lstAddItem** and section 3.2.2.7 **wm_lstInsertItem** descriptions for more details).

  funcTable:

  Pointer on a structure containing the comparison and the item destruction callbacks.

- Returned values

  This function returns a list pointer corresponding to the created list. This must be used in all further operations on this list.

3.2.2.3 The wm_lstDestroy Function

The wm_lstDestroy function allows to clear and then destroy the provided list.

- Prototype

```
void wm_lstDestroy ( wm_lst_t list );
```

  list:

  The list to destroy.

Note:

This function calls the **FreeItem** callback (if defined) on each item to delete it, before destroying the list.

3.2.2.4 The wm_lstClear Function

The **wm_lstClear** function allows to clear all the provided list items, without destroying the list itself (please refer to section 3.2.2.9 **wm_lstDeleteItem** function for notes on item deletion).

- Prototype

```
void wm_lstClear ( wm_lst_t list );
```

- Parameters

  list:

  The list to clear.

Note:

This function calls the **FreeItem** callback (if defined) on each item to delete it.

3.2.2.5 The wm_lstGetCount Function

The **wm_lstGetCount** function returns the current item count.

- Prototype

```
u16 wm_lstGetCount ( wm_lst_t list );
```

- Parameters

list:

The list from which to get the item count.

- Returned values

The number of items of the provided list. The function returns 0 if the list is empty.

3.2.2.6 The wm_lstAddItem Function

The **wm_lstAddItem** function allows to add an item to the provided list.

- Prototype

```
s16 wm_lstAddItem ( wm_lst_t    list
                    void *      item );
```

- Parameters

list:

The list to add an item to.

item:

The item to add to the list.

- Returned values

The position of the added item, or ERROR if an error occurred.

Notes:

- The **item** pointer should not point on a **const** or local buffer, as it is released in any item destruction operation.
- If the list has the WM_LIST_SORTED attribute, the item is inserted in the appropriate place after calling of the **CompareItem** callback (if defined). Otherwise, the item is appended at the end of the list.
- If the list has the WM_LIST_NODUPLICATES, the item is not inserted when the **CompareItem** callback (if defined) returns 0 on any previously added item. In this case, the returned index is the existing item index.

3.2.2.7 The wm_lstInsertItem Function

The **wm_lstInsertItem** function allows to insert an item to the provided list at the given location.

- Prototype

```
s16 wm_lstInsertItem ( wm_lst_t    list
                       void *      item
                       u16         index );
```

- Parameters

  list:

    The list to add an item to.

  item:

    The item to add to the list.

  index:

    The location where to add the item.

- Returned values

    The position of the added item, or **ERROR** if an error occured.

Notes:

- The item pointer should not point on a const or local buffer, as it is released in any item destruction operation.
- This function does not take list attributes into account and always inserts the provided item in the given index.

3.2.2.8 The wm_lstGetItem Function

The **wm_lstGetItem** function allows to read an item from the provided list, in the given index.

- Prototype

```
void * wm_lstGetItem ( wm_lst_t  list
                       u16       index );
```

- Parameters

  list:

    The list from which to get the item.

  index:

    The location where to get the item.

- Returned values

    A pointer on the requested item, or **NULL** if the index is not valid.

3.2.2.9 The wm_lstDeleteItem Function

The **wm_lstDeleteItem** function allows to delete an item of the provided list in the given indices.

- Prototype

```
s16 wm_lstDeleteItem ( wm_lst_t    list
                       u16         index );
```

- Parameters

list:

The list to delete an item from.

index:

The location where to delete the item.

- Returned values

The number of remaining items in the list, or **ERROR** if an error did occur.

Note:

This function calls the **FreeItem** callback (if defined) on the requested item to delete it.

The wm_lstFindItem Function

The **wm_lstFindItem** function allows to find an item in the provided list.

- Prototype

```
s16 wm_lstFindItem ( wm_lst_t    list
                     void *      item );
```

- Parameters

list:

The list where to search.

item:

The item to find.

- Returned values

The index of the found item if any, ERROR otherwise.

Note:

This function calls the **CompareItem** callback (if defined) on each list item, until it returns 0.

3.2.2.10 The wm_lstFindAllItem Function

The **wm_lstFindAllItem** function allows to find all items matching the provided one, in the given list.

- Prototype

```
s16 * wm_lstFindAllItem (wm_lst_t   list
                         void *     item );
```

- Parameters

list:

The list where to search.

item:

The item to find.

- Returned values

A s16 buffer containing the indices of all the items found, and ending with ERROR.

Important remark: This buffer should be released by the application when its processing is done.

Notes:

- This function calls the **CompareItem** callback (if defined) on each list item to get all those which match the provided item.
- This function should be used only if the list cannot be changed during the resulting buffer processing. Otherwise the **wm_lstFindNextItem** should be used.

3.2.2.11 The wm_lstFindNextItem Function

The **wm_lstFindNextItem** function allows to find the next item index of the given list, which corresponds with the provided one.

- Prototype

```
s16 wm_lstFindNextItem (wm_lst_t   list
                        void *     item );
```

- Parameters

list:

The list to search in.

item:

The item to find.

- Returned values

The index of the next found item if any, otherwise ERROR.

Note:

This function calls the **CompareItem** callback (if defined) on each list item to get those which match with the provided item. It should be called until it returns **ERROR**, in order to get the index of all items corresponding to the provided one. The difference with the **wm_lstFindAllItem** function is that, even if the list is updated between two calls to **wm_lstFindNextItem**, the function does not return a previously found item. To restart a search with the **wm_lstFindNextItem,** the **wm_lstResetItem** should be called first.

3.2.2.12 The wm_lstResetItem Function

The **wm_lstResetItem** function allows to reset all previously found items by the **wm_lstFindNextItem** function.

- Prototype

```
void wm_lstResetItem ( wm_lst_t    list
                       void *      item );
```

- Parameters

list:

The list to search in.

item:

The item to search, in order to reset all previously found items.

Note:

This function calls the **CompareItem** callback (if defined) on each list item to get those which match with the provided one.

3.2.3 Standard Library

3.2.3.1 Standard C Function Set

The available standard APIs are defined below:

```
ascii * wm_strcpy      ( ascii * dst, ascii * src );
ascii * wm_strncpy     ( ascii * dst, ascii * src, u32 n );
ascii * wm_strcat      ( ascii * dst, ascii * src );
ascii * wm_strncat     ( ascii * dst, ascii * src, u32 n );
u32     wm_strlen      ( ascii * str );
s32     wm_strcmp      ( ascii * s1, ascii * s2 );
s32     wm_strncmp     ( ascii * s1, ascii * s2, u32 n );
s32     wm_stricmp     ( ascii * s1, ascii * s2 );
s32     wm_strnicmp    ( ascii * s1, ascii * s2, u32 n );
ascii * wm_memset      ( ascii * dst, ascii c, u32 n );
ascii * wm_memcpy      ( ascii * dst, ascii * src, u32 n );
s32     wm_memcmp      ( ascii * dst, ascii * src, u32 n );
ascii * wm_itoa        ( s32 a, ascii * szBuffer );
s32     wm_atoi        ( ascii * p );
u8      wm_sprintf     ( ascii * buffer, ascii * fmt, ... );
```

Important remark about GCC compiler:

When using GCC compiler, due to internal standard C library architecture, it is strongly not recommended to use the "%f" mode in the wm_sprintf function in order to convert a float variable to a string. This leads to an ARM exception (product reset).

A way around for this conversion is:

float MyFloat; // float to display

ascii MyString [100]; // destination string

s16 d,f;

d = (s16) MyFloat * 1000; // Decimal precision: 3 digits

f = (MyFLoat * 1000) - d; // Decimal precision: 3 digits

wm_sprintf (MyString, "%d.%03d", (s16)MyFloat, f); // Decimal precision: 3 digits

3.2.3.2 String Processing Function Set

Some string processing functions are also available in this standard API.

Note:

All the following functions result as an ARM exception if a requested **ascii *** parameter is NULL.

ascii wm_isascii (ascii *c*);

Returns c if it is an ascii character ('a'/'A' to 'z'/'Z'), 0 otherwise.

ascii wm_isdigit (ascii *c*);

Returns c if it is a digit character ('0' to '9'), 0 otherwise.

ascii wm_ishexa (ascii *c*);

Returns c if it is a hexadecimal character ('0' to '9', 'a'/'A' to 'f'/'F'), 0 otherwise.

bool wm_isnumstring (ascii * *string*);

Returns TRUE if string is a numeric one, FALSE otherwise.

bool wm_ishexastring (ascii * *string*);

Returns TRUE if string is a hexadecimal one, FALSE otherwise.

bool wm_isphonestring (ascii * *string*);

Returns TRUE if string is a valid phone number (national or international format), FALSE otherwise.

u32 wm_hexatoi (ascii * *src*, u16 *iLen*);

If src is a hexadecimal string, converts it to a returned u32 of the given length, and 0 otherwise. As an example: wm_hexatoi ("1A", 2) returns 26, wm_hexatoi ("1A", 1) returns 1

u8 * wm_hexatoibuf (u8 * *dst*, ascii * *src*);

If src is a hexadecimal string, converts it to an u8 * buffer and returns a pointer on dst, and NULL otherwise. As an example, wm_hexatoibuf (dst, "1F06") returns a 2 bytes buffer: 0x1F and 0x06

```
ascii * wm_itohexa  ( ascii * dst, u32 nb, u8 len );
```

Converts nb to a hexadecimal string of the given length and returns a pointer on dst. For example, wm_itohexa (dst, 0xD3, 2) returns "D3", wm_itohexa (dst, 0xD3, 4) returns "00D3".

```
ascii * wm_ibuftohexa     ( ascii * dst, u8 * src, u16 len );
```

Converts the u8 buffer src to a hexadecimal string of the given length and returns a pointer on dst. Example with the src buffer filled with 3 bytes (0x1A, 0x2B and 0x3C), wm_ibuftohexa (dst, src, 3) returns "1A2B3C").

```
u16     wm_strSwitch      ( const ascii * strTest, ... );
```

This function must be called with a list of strings parameters, ending with NULL. strTest is compared with each of these strings (on the length of each string, with no matter of the case), and returns the index (starting from 1) of the string which matches if any, 0 otherwise.
Example:
wm_strSwitch ("TEST match", "test", "no match", NULL") returns 1,
wm_strSwitch ("nomatch", "nomatch a", "nomatch b", NULL) returns 0.

```
ascii * wm_strRemoveCRLF ( ascii * dst, ascii * src, u16 size );
```

Copy in dst buffer the content of src buffer, removing CR (0x0D) and LF (0x0A) characters, from the given size, and returns a pointer on dst.

```
ascii * wm_strGetParameterString (ascii *         dst,
                                  const ascii *   src,
                                  u16             Position );
```

If src is a string formatted as an AT response (for example "+RESP: 1,2,3") or as an AT command (for example "AT+CMD=1,2,3"), the function copies the parameter at Position offset (starting from 1) if it is present in the dst buffer, and returns a pointer on dst. It returns NULL otherwise.
Example:
wm_strGetParameterString (dst, "+WIND: 4", 1) returns "4",
wm_strGetParameterString (dst, "+WIND: 5,1", 2) returns "1",
wm_strGetParameterString (dst, "AT+CMGL=\"ALL\"", 1) returns "ALL".

3.2.4 Sound API

3.2.4.1 The wm_sndTonePlay Function

This function allows a tone to be played on the current speaker or on the buzzer. Frequency, gain and duration can be specified.

- Prototype

```
s32  wm_sndTonePlay (  wm_snd_dest_e     Destination,
                        u16              Frequency,
                        u8               Duration,
                        u8               Gain );
```

- Parameters

Destination:

Destination of the requested tone to play: speaker or buzzer.

```
typedef enum           {
  WM_SND_DEST_BUZZER,
  WM_SND_DEST_SPEAKER,
  WM_SND_DEST_GSM             /* do not use */
} wm_snd_dest_e;
```

Frequency:

For speaker: range is 1 Hz to 3999 Hz.

For buzzer: range is 1 Hz to 50000 Hz.

Duration:

This parameter sets tone duration (in unit of 20 ms). Applicable parameter range: 0-255.

Remark: when <duration> = 0, the duration is set to 70ms +/- 5ms (according to 3GPP 23.014).

Gain:

This parameter sets the tone gain.

Range of values is from 0 to 15.

<gain>	Speaker (db)	Buzzer (db)
0	0	-0.25
1	-0.5	-0.5
2	-1	-1
3	-1.5	-1.5
4	-2	-2
5	-3	-3

<gain>	Speaker (db)	Buzzer (db)
6	-6	-6
7	-9	-9
8	-12	-12
9	-15	-15
10	-18	-18
11	-24	-24
12	-30	-30
13	-36	-40
14	-42	-infinite
15	-infinite	-infinite

- Returned values

  OK on success, or negative error value.

- Example:

  An example of playing tone:

```
wm_sndTonePlay ( WM_SND_DEST_BUZZER, 1000, 0, 9 );
```

3.2.4.2 The wm_sndTonePlayExt Function

This function allows a dual tone (two frequencies) to be played on the specified output. Frequencies, gains and duration can be specified.

Note:

Only the speaker output is able to play tones in two frequencies. The second tone parameters are ignored on the buzzer output.

- Prototype

```
s32  wm_sndTonePlayExt (wm_snd_dest_e    Destination,
                         u16              Frequency,
                         u16              Frequency2,
                         u8               Duration,
                         u8               Gain,
                         u8               Gain2 );
```

- Parameters

Destination:

Destination of the requested tone to play: speaker or buzzer.

```
typedef enum
{
  WM_SND_DEST_BUZZER,
  WM_SND_DEST_SPEAKER,
  WM_SND_DEST_GSM             /* do not use */
} wm_snd_dest_e;
```

Frequency, Frequency2:

For speaker: range is from 1 Hz to 3999 Hz.

For buzzer: range is from 1 Hz to 50000 Hz.

Please remember that the Frequency2 parameter is only processed on the speaker output.

Duration:

This parameter sets tone duration (in unit of 20 ms). Applicable parameter range: 0-255.

Remark: when <duration> = 0, the duration is set to 70ms +/- 5ms (according to 3GPP 23.014).

Gain, Gain2:

This parameter sets the tones gain. Gain parameter applies to Frequency value, and Gain2 applies to the Frequency2 one.

Range of values is from 0 to 15.

<gain>	Speaker (db)	Buzzer (db)
0	0	-0.25
1	-0.5	-0.5
2	-1	-1
3	-1.5	-1.5
4	-2	-2
5	-3	-3
6	-6	-6
7	-9	-9
8	-12	-12
9	-15	-15

<gain>	Speaker (db)	Buzzer (db)
10	-18	-18
11	-24	-24
12	-30	-30
13	-36	-40
14	-42	-infinite
15	-infinite	-infinite

- Returned values

  OK on success, or a negative error value

- Example:

  An example of playing tone:

```
wm_sndTonePlayExt ( WM_SND_DEST_SPEAKER, 1000, 2000, 0, 9, 10 );
```

3.2.4.3 The wm_sndToneStop Function

This function stops playing a tone on the current speaker or on the buzzer.

- Prototype

```
s32  wm_sndToneStop ( wm_snd_dest_e Destination );
```

- Parameters

  Destination:

  Destination of the current playing tone to stop: speaker or buzzer.

- Returned values

  OK on success, or a negative error value.

- Example:

  An example of stopping tone:

```
wm_sndToneStop ( WM_SND_DEST_BUZZER );
```

3.2.4.4 The wm_sndDtmfPlay Function

This function allows a DTMF tone to be played on the current speaker or over the GSM network (in communication only). DTMF, gain (only for speaker) and duration can be specified.

Remark: It is not possible to play DTMF on the buzzer.

- Prototype

```
s32  wm_sndDtmfPlay (  wm_snd_dest_e      Destination,
                        ascii              Dtmf,
                        u8                 Duration,
                        u8                 Gain );
```

- Parameters

Destination:

Destination of the requested DTMF tone to play: speaker or/and over the GSM network (in communication only).

```
typedef enum            {
  WM_SND_DEST_BUZZER,       /* do not use */
  WM_SND_DEST_SPEAKER,
  WM_SND_DEST_GSM
} wm_snd_dest_e;
```

Dtmf:

Value must be in { ‘0’ - ‘9’, ’*’, ’#’, ’A’, ’B’, ’C’, ’D’ }

Duration:

This parameter sets tone duration (in unit of 20 ms). Applicable parameter range: 0-255.

Remark: when <duration> = 0, the duration is set to 70ms +/- 5ms (according to 3GPP 23.014).

Gain:

Only for speaker.

This parameter sets the tone gain.

Range of values is from 0 to 15.

- Returned values

OK on success, or a negative error value

- Example:

An example of playing DTMF:

```
wm_sndDtmfPlay ( WM_SND_DEST_SPEAKER, ‘A’, 100, 9 );
```


3.2.4.5 The wm_sndDtmfStop Function

This function stops playing a dtmf on the current speaker or over the GSM network (in communication only).

- Prototype

```
s32  wm_sndDtmfStop ( wm_snd_dest_e Destination );
```

- Parameters

  Destination:

  Destination of the current playing tone to stop, this must be a speaker (GSM network DTMF cannot be stopped).

- Returned values

  OK on success, or a negative error value

- Example:

  An example of stopping DTMF:

```
wm_sndDtmfStop ( WM_SND_DEST_SPEAKER );
```

3.2.4.6 The wm_sndMelodyPlay Function

This function plays a melody. **Destination**, **Melody**, **Tempo**, **Cycle** and **Gain** can be specified.

- Prototype

```
s32  wm_melody_play (   wm_snd_dest_e      Destination,
                        u16*               Melody,
                        u16                Tempo,
                        u8                 Cycle,
                        u8                 Gain );
```

- Parameters

  Destination:

  Destination of the melody to play: speaker or buzzer.

```
typedef enum            {
      WM_SND_DEST_BUZZER,
      WM_SND_DEST_SPEAKER,
      WM_SND_DEST_GSM         /* do not use */
} wm_snd_dest_e;
```

  Melody:

  Melody to play. A melody is defined by an u16 table, where each element defines a note event, with duration and sound definition.

```
// Melody sample
const u16 MyMelody [ ]=
{
  WM_SND_E1 |  WM_SND_QUAVER,
  WM_SND_F1 |  WM_SND_MBLACK,
  WM_SND_G6S |  WM_SND_QUAVER,
};
```

```
typedef enum {
        WM_SND_C0 ,         // C0
        WM_SND_C0S ,        // C0#
        WM_SND_D0 ,         // D0
        WM_SND_D0S ,        // D0#
        WM_SND_E0 ,         // E0
        WM_SND_F0 ,         // F0
        WM_SND_F0S ,        // F0#
        WM_SND_G0 ,         // G0
        WM_SND_G0S ,        // G0#
        WM_SND_A0 ,         // A0
        WM_SND_A0S ,        // A0#
        WM_SND_B0 ,         // B0
        WM_SND_C1 ,         // C1
…
        WM_SND_NO_SOUND=0xFF
} wm_sndNote_e;

#define WM_SND_ROUND           0x1000
#define WM_SND_MWHITEP         0x0C00
#define WM_SND_MWHITE          0x0800
#define WM_SND_MBLACKP         0x0600
#define WM_SND_MBLACK          0x0400
#define WM_SND_QUAVERP         0x0300
#define WM_SND_QUAVER          0x0200
#define WM_SND_MSHORT          0x0100
```

Tempo:

Tempo to apply (duration a black x 20 ms).

Cycle:

Number of times that the melody should be played (0 = infinite)

Gain:

Volume to apply, range of values is 0 to 15.

- Returned values

  OK on success, or a negative error value

- Example:

  An example of playing melody:

```
MelodyPlay ( WM_SND_DEST_SPEAKER, MyMelody, 6, 1, 9 );
```

3.2.4.7 The wm_sndMelodyStop Function

This function stops playing a melody on the current speaker or on the buzzer.

- Prototype

  `s32 wm_sndMelodyStop ( wm_snd_dest_e Destination );`

- Parameters

  Destination:

  Destination of the current playing melody to stop: speaker or buzzer.

- Returned values

  OK on success, or a negative error value

- Example:

  An example of stopping a melody:

```
wm_sndMelodyStop ( WM_SND_DEST_SPEAKER );
```

3.3 AT Commands Service

3.3.1 Required Header File

The header file for the functions dealing with AT commands is:

adl_at.h

3.3.2 Unsolicited Responses

An unsolicited response is a string sent by the Wavecom Firmware to applications in order to provide them unsolicited event information (ie. not in response to an AT command).

ADL applications may subscribe to an unsolicited response in order to receive the event in the provided handler.

Once an application has subscribed to an unsolicited response, it will have to unsubscribe from it to stop the callback function being executed every time the matching unsolicited response is sent from the Wavecom Firmware.

Multiple subscriptions: Each unsolicited response may be subscribed several times. If an application subscribes to an unsolicited response with handler 1 and then subscribes to the same unsolicited response with handler 2, every time the ADL parser receives this unsolicited response handler 1 and then handler 2 will be executed.

3.3.2.1 The adl_atUnSoSubscribe Function

This function subscribes to a specific unsolicited response with an associated callback function: when the required unsolicited response is sent from the Wavecom Firmware, the callback function will be executed.

- Prototype

```
s16 adl_atUnSoSubscribe ( ascii *                UnSostr,
                          adl_atUnSoHandler_t   UnSohdl )
```

- Parameters

  UnSostr:

  The name (as a string) of the unsolicited response we want to subscribe to. This parameter can also be set as an **adl_rspID_e** response ID. Please refer to § 3.21 for more information.

  UnSohdl:

  A handler to the callback function associated to the unsolicited response.

  The callback function is defined as follow:

```
typedef bool (* adl_atUnSoHandler_t) (adl_atUnsolicited_t *)
```

The argument of the callback function will be a '**adl_atUnsolicited_t**' structure, holding the unsolicited response we subscribed to.

The 'adl_atUnsolicited_t' structure defined as follow (it is declared in the adl_at.h header file):

```
typedef struct
{
  adl_strID_e RspID;          // Standard response ID
  adl_atPort_e Dest;          // Unsolicited response destination port
  u16 StrLength;              /* the length of the string (name) of the
                              unsolicited response */
  ascii StrData[1];           /* a pointer to the string (name) of the
                              unsolicited response */
} adl_atUnsolicited_t;
```

The RspID field is the parsed standard response ID if the received response is a standard one. Refer to § 3.21 for more information.

The Dest field is the unsolicited response original destination port. If it is set to ADL_PORT_NONE, unsolicited response is required to be broadcasted on all ports.

The return value of the callback function will have to be TRUE if the unsolicited string is to be sent to the external application (on the port indicated by the Dest field, if not set to ADL_PORT_NONE, otherwise on all ports), and FALSE otherwise.

Note:

That in case of several handlers associated to the same unsolicited response, all of them have to return TRUE for the unsolicited response to be sent to the external application.

- Returned values
  - **OK** on success
  - **ERROR** if an error occurred.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.3.2.2 The adl_atUnSoUnSubscribe Function

This function unsubscribes from an unsolicited response and its handler.

- Prototype

```
s16 adl_atUnSoUnSubscribe ( ascii *              UnSostr,
                            adl_atUnSoHandler_t UnSohdl )
```

- Parameters

  UnSostr:

  The string of the unsolicited response we want to unsubscribe to.

  UnSohdl:

  The callback function associated to the unsolicited response.

- Returned values
  - **OK** if the unsolicited response was found.
  - **ERROR** otherwise.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context)

- Example

```
/* callback function */
bool Wind4_Handler(adl_atUnsolicited_t *paras)
{
    /* Unsubscribe to the '+WIND: 4' unsolicited response */
    adl_atUnSoUnSubscribe("+WIND: 4",
                     (adl_atUnSoHandler_t)Wind4_Handler);
    adl_atSendResponse(ADL_AT_RSP, "\r\nWe have received a Wind 4\r\n");
    /* We want this response to be sent to the external application,
    * so we return TRUE */
    return TRUE;
}

/*main function */
void adl_main(adl_InitType_e adlInitType)
{
    /* Subscribe to the '+WIND: 4' unsolicited response */
    adl_atUnSoSubscribe("+WIND: 4",
                     (adl_atUnSoHandler_t)Wind4_Handler);
}
```

3.3.3 Responses

3.3.3.1 The adl_atSendResponse function

This function sends the provided text to any external application connected to the required port, as a response, an unsolicited response or an intermediate response, according to the requested type.

- Prototype

```
s32 adl_atSendResponse (u16        Type,
                        ascii *    String )
```

- Parameters

Type:

This parameter is composed of the response type, and the destination port where to send the response. The type & destination combination has to be done with the following macro:

ADL_AT_PORT_TYPE (_port, _type)

The _port argument has to be a defined value of the adl_atPort_e type, and this required port has to be available (cf. the AT/FCM port Service) ; sending a response on an Open AT® the GSM or GPRS based port will have no effects).

Note:

With the **ADL_AT_UNS** type value, if the **ADL_AT_PORT_TYPE** macro is not used, the unsolicited response will be broadcasted on all currently opened ports.

If the **ADL_AT_PORT_TYPE** macro is not used with the **ADL_AT_RSP** & **ADL_AT_INT** types, responses will be by default sent on the UART 1 port. If this port is not opened, responses will not be displayed.

The _type argument has to be one of the values defined below:

- **ADL_AT_RSP:**
  Terminal response (have to ends an incoming AT command).
  A destination port has to be specified.
  Sending such a response will flush all previously buffered unsolicited responses on the required port.

- **ADL_AT_INT:**
  Intermediate response (text to display while an incoming AT command is running).
  A destination port has to be specified.
  Sending such a response will just display the required text, without flushing all previously buffered unsolicited responses on the required port.

- **ADL_AT_UNS**:
  Unsolicited response (text to be displayed out of a currently running command process).
  For the required port (if any) or for each currently opened port (if the **ADL_AT_PORT_TYPE** macro is not used), if an AT command is currently running (ie. the command was sent by the external application, but this command answer has not be sent back yet), any unsolicited response will automatically be buffered, until a terminal response is sent on this port.

String:

The text to be sent.

Please note that this is exactly the text string to be displayed on the required port (ie. all carriage return & line feed characters ("\r\n" in C language) have to be sent by the application itself).

- Returned values
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).
  - **OK** if the function is successfully executed.

3.3.3.2 The adl_atSendStdResponse Function

This function sends the provided standard response to the required port, as a response, an unsolicited response or an intermediate response, according to the requested type.

- Prototype

```
s32 adl_atSendStdResponse (u8            Type,
                           adl_strID_e   RspID )
```

- Parameters

Type:

Same use as the `adl_atSendResponse` Type parameter.

RspID:

Standard response ID to be sent (see 3.21 for more information).

- Returned values
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).
  - **OK** if the function is successfully executed.

3.3.3.3 The adl_atSendStdResponseExt Function

This function sends the provided standard response with an argument to the required port, as a response, an unsolicited response or an intermediate response, according to the requested type.

- Prototype

```
s32 adl_atSendStdResponseExt (u8             Type,
                              adl_strID_e   RspID,
                              u32            arg )
```

- Parameters

  Type:

  Same use as the `adl_atSendResponse` Type parameter.

  RspID:

  Standard response ID to be sent (see 3.21for more information).

  arg:

  Standard response argument. According to response ID, this argument should be an **u32** integer, or an **ascii *** string.

- Returned values
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).
  - o **OK** if the function is successfully executed.

3.3.3.4 Additional Macros for Specific Port Access

The above Response sending functions may be also used with the macros below, which provide the additional Port argument: it should avoid heavy code including each time the **ADL_AT_PORT_TYPE** macro call.

```
#define adl_atSendResponsePort(_t,_p,_r)
      adl_atSendResponse(ADL_AT_PORT_TYPE(_p,_t),_r)
#define adl_atSendStdResponsePort(_t,_p,_r)
      adl_atSendStdResponse(ADL_AT_PORT_TYPE(_p,_t),_r)
#define adl_atSendStdResponseExtPort(_t,_p,_r,_a)
      adl_atSendStdResponseExt(ADL_AT_PORT_TYPE(_p,_t),_r,_a)
```

3.3.4 Incoming AT Commands

An ADL application may subscribes to an AT command string, in order to receive events each time an external application sends this AT command on one of the Wireless CPU®s ports.

Once the application has subscribed to a command, it will have to unsubscribe to stop the callback function being executed every time this command is sent by an external application.

<u>Multiple subscriptions</u>: if an application subscribes to a command with a handler and subscribes then to the same command with another handler, every time this

command is sent by the external application both handlers will be successfully executed (in the subscription order).

Important note about incoming concatenated command:

ADL is able to recognize and process concatenated commands coming from external applications (Please refer to AT Commands Interface Guide (document [2]) for more information on concatenated commands syntax).

In this case, this port enters a specific concatenation processing mode, which will end as soon as the last command replies **OK**, or if one of the used command replies an ERROR code. During this specific mode, all other external command requests will be refused on this port: any external application connected to this port will receive a "+CME ERROR: 515" code if it tries to send another command. The embedded application can continue using this port for its specific processes, but it has to be careful to send one (at least one, and only one) terminal response for each subscribed command.

If a subscribed command is used in a concatenated command string, the corresponding handler will be notified as if the command was used alone.

In order to handle properly the concatenation mechanism, each subscribed command has to finally answer with a single terminal response (**ADL_STR_OK, ADL_STR_ERROR** or other ones), otherwise the port will stay in concatenation processing mode, refusing all internal and external commands on this one.

The defined operations are:

- A **adl_atCmdSubscribeExt** function to subscribe to a command with providing a Context.
- A **adl_atCmdSubscribe** function to subscribe to a command without providing a Context.
- A **adl_atCmdUnSubscribe** function to unsubscribe to a command.

3.3.4.1 Required Header File

The required header file is:

adl_CmdHandler.h

3.3.4.2 The adl_atCmdPreParser_t Structure

This structure contains information about AT command.

- Code

```
typedef struct
{
    u16             Type;       // Type
    u8              NbPara;     // Number of parameters
    adl_atPort_e    Port;       // Port
    wm_lst_t        ParaList;   // List of parameters
    u16             StrLength;  // Incoming command length
    u16             NI         // Notification Identifier
    void *          Contxt     // Context
    ascii           StrData[1]; // Incoming command address
} adl_atCmdPreParser_t;
```

- Description

Type

Incoming command type (will be one of the required ones at subscription time), detected by the ADL pre-processing.

NbPara

Non NULL parameters number (if Type is **ADL_CMD_TYPE_PARA**), or 0 (with other type values).

Port:

Port on which the command was sent by the external application.

ParaList:

Only if Type is **ADL_CMD_TYPE_PARA**. Each parameter may be accessed by the **ADL_GET_PARAM(_p,_i)** macro. If a string parameter is provided (eg. AT+MYCMD="string"), the quotes will be removed from the returned string (eg. **ADL_GET_PARAM(para,0)** will return "string" (without quotes) in this case). If a parameter is not provided (eg. AT+MYCMD), the matching list element will be set to NULL (eg. **ADL_GET_PARAM(para,0)** will return NULL in this case).

StrLength:

Incoming command string buffer length.

NI:

This parameter is to hold the Notification Identifier provided by the command handler when re sending the command already subscribed to solve any loop effect.

Contxt:

A context holding information gathered at the time the command is subscribed (if provided).

StrData[1]:

Incoming command string buffer address. If the incoming command from the external application is containing useless spaces (" ") or semi-colon (";") characters, those will automatically be removed from the command string (e.g. if an external application sends "AT+MY CMD;" string, the command handler will receive "AT+MYCMD").

3.3.4.3 The adl_ atCmdSubscriptionPort_e Type

Basic required subscription port affected.

- Code

```
typedef enum
{
          ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT,
          ADL_CMD_SUBSCRIPTION_ALL_PORTS
} adl_atCmdSubscriptionPort_e;
```

- Description

ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT: The subscription is only concerning command received on the external port.

ADL_CMD_SUBSCRIPTION_ALL_PORTS: The subscription is concerning command received on all ports.if an application subscribes to a command with a handler and subscribes then to the same command with another handler, every time this command is sent by the external application both handlers will be successively executed (in the subscription order).

Caution:

In this current release **ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT** is the only valid choice

3.3.4.4 ADL_GET_PARAM

Macro to get the requested parameter.

- Code

```
#define ADL_GET_PARAM  (_P_,
                          _i_)((ascii*)wm_lstGetIitem(_P_->ParaList,_i_))
```

- Parameters

_P_:

command handler parameter (refer to **adl_atCmdPreParser_t** structure about pointer to use).

_i_:

parameter index from 0 to NbPara (refer to **adl_atCmdPreParser_t** structure for more information about NbPara).

3.3.4.5 The adl_atCmdHandler_t Command Handler

Such a call-back function has to be supplied to ADL through the **adl_atCmdSubscribe** interface in order to process AT command subscribed.

- Prototype

```
typedef void (*) adl_atCmdHandler_t (adl_atCmdPreParser_t *Params)
```

- Parameters

Params:

Contains information about AT response (refer to **adl_atCmdPreParser_t** for more information).

Note:

The command handler has the responsability to send unsollicited/intermediate reponses and at least one terminal response.

3.3.4.6 The adl_atCmdSubscribe Function

This function subscribes to a specific command with an associated callback function, so that next time the required command is sent exclusively by an external application, the callback function will be executed.

- Prototype

```
s16 adl_atCmdSubscribe (ascii *             Cmdstr,
                        adl_atCmdHandler_t  Cmdhdl,
                        u16                 Cmdopt )
```

- Parameters

Cmdstr:

The string (name) of the command we want to subscribe to. Since this service only handles AT commands, this string has to begin by the "AT" characters.

Cmdhdl:

The handler of the callback function associated to the command. (Refer to **adl_atCmdHandler_t** for more information about callback function).

Cmdopt:

This flag combines with a bitwise 'OR' ('|' in C language) the following information:

Command type	Value	Meaning
ADL_CMD_TYPE_PARA	0x0100	'AT+cmd=x, y'is allowed. The execution of the callback function also depends on whether the number of argument is valid or not. Information about number of arguments is combined with a bitwise 'OR' : **ADL_CMD_TYPE_PARA | 0xXY** , where X which defines maximum argument number for incoming command and Y which defines minimum argument number for incoming command..
ADL_CMD_TYPE_TEST	0x0200	'AT+cmd=?' is allowed.
ADL_CMD_TYPE_READ	0x0400	'AT+cmd?' is allowed.
ADL_CMD_TYPE_ACT	0x0800	'AT+cmd' is allowed.
ADL_CMD_TYPE_ROOT	0x1000	All commands starting with the subscribed string are allowed but without the ending character ";" which is parsed for concatenated commands mode. The handler will only receive the whole AT string (no parameters detection). For example, if the "at-" string is subscribed, all "at-cmd1", "at-cmd2", etc. strings will be received by the handler, however the only string "at-" is not received.
ADL_CMD_TYPE_ROOT_EXT	0x2000	All commands starting with the subscribed string are allowed even with the ending character ";" this means that such a command will not be usable in a concatenated AT commands string. The handler will only receive the whole AT string (no parameters detection). For example, if the "at-" string is subscribed, all "at-cmd1", "at-cmd2", etc. strings will be received by the handler, however the only string "at-" is not received. Note: in this current release **ADL_CMD_TYPE_ROOT_EXT** is behaving like **ADL_CMD_TYPE_ROOT**

Note:

If **ADL_CMD_TYPE_ROOT_EXT** is associated with others it has priority and therefore the command cannot be recognized as a concatenated one.

In this current release **ADL_CMD_TYPE_ROOT_EXT** behaving like **ADL_CMD_TYPE_ROOT.**

- Returned values
  - **OK** on success.
  - **ERROR** if an error occurred.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.3.4.7 The adl_atCmdSubscribeExt Function

This function subscribes to a specific command with an associated callback function, so that next time the required command is sent by an external application or on all ports (depending on the Cmdport parameter), the callback function will be executed.

- Prototype

```
s16 adl_atCmdSubscribeExt (ascii *                        Cmdstr,
                           adl_atCmdHandler_t             Cmdhdl,
                           u16                            Cmdopt,
                           void *                         Contxt,
                           adl_atCmdSubscriptionPort_e    Cmdport )
```

- Parameters

Cmdstr:

The string (name) of the command we want to subscribe to. Since this service only handles AT commands, this string has to begin by the "AT" characters.

Cmdhdl:

The handler of the callback function associated to the command. (Refer to **adl_atCmdHandler_t** for more information about callback function).

Cmdopt:

This flag combines with a bitwise 'OR' ('|' in C language) the following information:

Command type	Value	Meaning
ADL_CMD_TYPE_PARA	0x0100	'AT+cmd=x, y'is allowed. The execution of the callback function also depends on whether the number of argument is valid or not. Information about number of arguments is combined with a bitwise 'OR' : ADL_CMD_TYPE_PARA | 0xXY , where X which defines maximum argument number for incoming command and Y which defines minimum argument number for incoming command..
ADL_CMD_TYPE_TEST	0x0200	'AT+cmd=?' is allowed.
ADL_CMD_TYPE_READ	0x0400	'AT+cmd?' is allowed.

Command type	Value	Meaning
ADL_CMD_TYPE_ACT	0x0800	'AT+cmd' is allowed.
ADL_CMD_TYPE_ROOT	0x1000	All commands starting with the subscribed string are allowed but without the ending character ";" which is parsed for concatenated commands mode. The handler will only receive the whole AT string (no parameters detection). For example, if the "at-" string is subscribed, all "at-cmd1", "at-cmd2", etc. strings will be received by the handler, however the only string "at-" is not received.
ADL_CMD_TYPE_ROOT_EXT	0x2000	All commands starting with the subscribed string are allowed even with the ending character ";" this means that such a command will not be usable in a concatenated AT commands string. The handler will only receive the whole AT string (no parameters detection). For example, if the "at-" string is subscribed, all "at-cmd1", "at-cmd2", etc. strings will be received by the handler, however the only string "at-" is not received. Note: In this current release **ADL_CMD_TYPE_ROOT_EXT** is behaving like **ADL_CMD_TYPE_ROOT**

Note:

If **ADL_CMD_TYPE_ROOT_EXT** is associated with others it has priority and therefore the command cannot be recognized as a concatenated one.

Caution:

In this current release **ADL_CMD_TYPE_ROOT_EXT** is behaving like **ADL_CMD_TYPE_ROOT**

Contxt:

Context made to hold information gathered at the time the command is subscribed.

Cmdport:

Port on which the command is subscribed (type of to **adl_atCmdSubscriptionPort_e**).

ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT

ADL_CMD_SUBSCRIPTION_ALL_PORTS

Note:

In this current release **ADL_CMD_SUBSCRIPTION_ONLY_EXTERNAL_PORT** is the only valid choice

- Returned values
  - **OK** on success.
  - **ERROR** if an error occurred.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.3.4.8 The adl_atCmdUnSubscribe Function

This function unsubscribes from a command and its handler.

- Prototype

```
s16 adl_atCmdUnSubscribe (ascii *            Cmdstr,
                          adl_atCmdHandler_t Cmdhdl )
```

- Parameters

Cmdstr:

The string (name) of the command we want to unsubscribe from.

Cmdhdl:

The handler of the callback function associated to the command.

- Returned values
  - **OK** on success,
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.
  - **ERROR** otherwise.

3.3.4.9 The adl_atCmdSetQuietMode Function

This function allows to set Quiet mode. In this mode, terminal responses are not send. This function has the same behaviour as ATQ command behaviour.

- Prototype

```
void adl_atCmdSetQuietMode (bool   IsQuiet )
```

- Parameters

IsQuiet:

Quiet mode setting:

- TRUE: Quiet mode is activated
- FALSE: Quiet mode is deactivated. Default value.

3.3.4.10 Example

This example demonstrates how to use the AT Command Subscription/Unsubscriptions service in a nominal case (error cases not handled) with a Wireless CPU®.

Complete examples using the AT Command service are also available on the SDK.

```
// ati callback function
    void ATI_Handler(adl_atCmdPreParser_t *paras)
    {
        // we send a terminal response
        adl_atSendStdResponsePort(ADL_AT_RSP, paras->Port, ADL_STR_OK);
    }

    // function 2
    void function2(adl_InitType_e adlInitType)
    {
        // We unsubscribe the command ;
        adl_atCmdUnSubscribe("ati",
                             (adl_atCmdHandler_t)ATI_Handler);
    }

    // function 1
    void function1(adl_InitType_e adlInitType)
    {
        // Subscribe to the 'ati' command.
        adl_atCmdSubscribe("ati",
                           (adl_atCmdHandler_t)ATI_Handler,
                           ADL_CMD_TYPE_ACT);
    }
```

3.3.5 Outgoing AT Commands

The following functions allow to send a command on the required port and allows the subscription to several responses and intermediate responses with one associated callback function, so that when any of the responses or intermediate responses we subscribe to will be received by the ADL parser, the callback function will be executed.

The defined operations are:

- **adl_atCmdCreate** function to send a command on the required port and allow the subscription to several responses and intermediate responses with one associated callback function, so that when any of the responses or intermediate responses we subscribe to will be received by the ADL parser, the callback function will be executed.
- **adl_atCmdSend** same function as **adl_atCmdCreate** without the rspflag argument and instead sending the command to the Open AT internal port.
- **adl_atCmdSendExt** same function as adl_atCmdCreate() allowing the usage of the Notification Identifier (see Note 3 below).
- **adl_atCmdSendText** function to allow to provide a running "Text Mode" command on a specific port (e.g. "AT+CMGW") with the required text. This function has to be used as soon as the prompt response ("> ") comes in the response handler provided on **adl_atCmdCreate**/**adl_atCmdSend**/ **adl_atCmdSendExt** function call.

Note:

In this current release the notification identifier (NI) is not used.

3.3.5.1 Required Header File

The header file is:

adl_CmdStackHandler.h

3.3.5.2 The adl_atResponse_t Structure

This structure contains information about AT command.

- code

```
typedef struct
{
    adl_strID_e     RspID;        // RspID
    adl_atPort_e    Dest;         // Dest
    u16             StrLength;    // Response length
    void *          Contxt;       // Context
    bool            IsTerminal;   // Terminal response flag
    u8              Pad [3];      //  Reserved for future use
    ascii           StrData[1];   // Response address
} adl_atResponse_t;
```

- Description

RspID:

Detected standard response ID if the received response is a standard one.

Dest:

Port on which the command has been executed; it is also the destination port where the response will be forwarded if the handler returns TRUE.

StrLength:

Response string buffer length.

Contxt:

A context holding information gathered at the time the command is sent (if provided).

IsTerminal:

A boolean flag indicating if the received response is the terminal one (TRUE) or an intermediate one (FALSE).

StrData[1]:

Response string buffer address.

3.3.5.3 The adl_atRspHandler_t

Such a call-back function has to be supplied to ADL through the **adl_atCmdCreate**/ **adl_atCmdSend**/**adl_atCmdSendExt** interface in order to process AT response subscribed.

- Prototype

```
typedef bool(*) adl_atRspHandler_t (adl_atResponse_t *Params)
```

- Parameters

Params:

Contains information about AT response (refer to **adl_atResponse_t** for more information).

- Returned value

The return value of the callback function has to be TRUE if the response string has to be sent to the provided port, FALSE otherwise.

3.3.5.4 The adl_atCmdCreate Function

Add command to the required port command stack, in order to be executed as soon as this port is ready.

- Prototype

```
s8 adl_atCmdCreate (   ascii *              atstr,
                       u16                  rspflag,
                       adl_atRspHandler_t   rsphdl,
                                            … )
```

- Parameters

atstr:

The string (name) of the command we want to send. Since this service only handles AT commands, this string has to begin by the "AT" characters.

rspflag:

This parameter is composed of the unsubscribed responses destination flag, and the port where to send the command. The flag & destination combination has to be done with the following macro:

`ADL_AT_PORT_TYPE ( _port, _flag )`

- The `_port` argument has to be a defined value of the `adl_atPort_e` type, and this required port has to be available (cf. the AT/FCM port Service). If this port is not available, or if it is a GSM or GPRS based one, the command will not be executed.
- The `_flag` argument has to be one of the values defined below:
  - If set to TRUE: the responses and intermediate responses of the sent command that are not subscribed (ie. not listed in the **`adl_atCmdCreate`** function arguments) will be sent on the required port.
  - If set to FALSE they will not be sent to the external application.
- If the **`ADL_AT_PORT_TYPE`** macro is not used, by default the command will be sent to the Open AT® virtual port (see next paragraph for more information about AT commands ports).

rsphdl:

The response handler of the callback function associated to the command.

...:

A list of strings of the response to subscribed to. This list has to be terminated by NULL.

- Returned values
  - **`OK`** on success
  - **`ERROR`** if an error occurred
  - **`ADL_RET_ERR_SERVICE_LOCKED`** if called from a low level interrupt handler

Note:

Arguments **`rsphdl`** and the list of subscribed responses can be set to NULL to only send the command.

3.3.5.5 The adl_atCmdSend Function

Add command to the required port command stack, in order to be executed as soon as this port is ready.

- Prototype

```
s8 adl_atCmdSend     (  ascii *            atstr,
                        adl_atRspHandler_t rsphdl,
                                           … )
```

- Parameters

atstr:

The string (name) of the command we want to send. Since this service only handles AT commands, this string has to begin by the "AT" characters.

rsphdl:

The response handler of the callback function associated to the command.

...:

A list of strings of the response to subscribed to. This list has to be terminated by NULL.

- Returned values
  - **OK** on success
  - **ERROR** if an error occurred
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler

Note:

Arguments rsphdl and the list of subscribed responses can be set to NULL to only send the command

3.3.5.6 The adl_atCmdSendExt Function

This function sends AT command with 2 added arguments compared to **adl_atCmdCreate / adl_atCmdSend** : a NI (Notification Identifier) and a Context.

Add command to the required port command stack, in order to be executed as soon as this port is ready.

- Prototype

```
s8 adl_atCmdSendExt   ( ascii *            atstr,
                        adl_atPort_e       port,
                        u16                NI,
                        ascii *            Contxt,
                        adl_atRspHandler_t rsphdl,
                                           … )
```

- Parameters

  atstr:

  The string (name) of the command we want to send. Since this service only handles AT commands, this string has to begin by the "AT" characters.

  port:

  The required port on which the command will be executed.

  NI:

  This parameter is to hold the Notification Identifier provided by the command handler when re sending the command already subscribed to solve any loop effect.

Note:

In this current release the notification identifier (NI) is not used.

  Contxt:

  Context made to hold information gathered at the time the command was sent.

  rsphdl:

  The response handler of the callback function associated to the command.

  ...:

  A list of strings of the response to subscribed to. This list has to be terminated by NULL.

- Returned values
  - **OK** on success
  - **ERROR** if an error occurred
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler

Note:

Arguments rsphdl and the list of subscribed responses can be set to NULL to only send the command.

3.3.5.7 The adl_atCmdSendText Function

Sends text for a running text command.

- Prototype

```
s8 adl_atCmdSendText ( adl_port_e  Port,
                       ascii *     Text )
```

- Parameters

  Port:

  Port on which is currently running the "Text Mode" command, waiting for some text input.

Text:

Text to be provided to the running "Text Mode" command on the required port. If the text does not end with a 'Ctrl-Z' character (0x1A code), the function will add it automatically.

- Returned values
  - o **OK** on success
  - o **ERROR** if an error occurred
  - o **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level Interrupt handler.
- Example

This example demonstrates how to use the AT Command Sending service in a nominal case (error cases not handled) with a Wireless CPU®.

Complete examples using the AT Command service are also available on the SDK.

- Example 1

```
// ati responses callback function
    s16 ATI_Response_Handler(adl_atResponse_t *paras)
    {
       TRACE((1, "Reponse handled"));
       TRACE((1, paras->StrData));
       return FALSE;
    }

    // function 1
    void function1(adl_InitType_e adlInitType)
    {
        // We send ati and subscribe to its responses
        adl_atCmdSend("ati",
                      (adl_atRspHandler_t)ATI_Response_Handler,
                      "*",
                      NULL);
    }
```

- Example 2

```
// at+mycmd responses callback function 2
    s16 AT_MYCMD_Response_Handler_2(adl_atResponse_t *paras)
    {
        // we send a terminal response
       adl_atSendStdResponsePort(ADL_AT_RPS, paras-> Port, ADL_STR_OK);
       return FALSE;
    }

    // at+mycmd callback function 1
    void AT_MYCMD_Handler_l(adl_atCmdPrepaerser_t *paras)
    {
        // we send a terminal response
        adl_atSendStdResponsePort(ADL_AT_RSP, paras->Port, ADL_STR_OK);

    }
// at+mycmd callback function 2
void AT_MYCMD_Handler_2(adl_atCmdPreParser_t *paras)
}
        // Only spying we resend the command
        adl_atCmdSendExt(paras->StrData,
                         paras->Port,
                         0,
                         NULL,
                         (adl_atRspHandler_t)AT_MYCMD_Response_Handler_2
                          "*",
                         NULL);
}

// function 1
void function1(adl_InitType_e adl InitType)
{
        // Subscribe to the 'at+mycmd' command.
        adl_atCmdSubscribe ("attmycmd",
                           (adl_atCmdHandler_t)AT_MYCMD_Handler_1
                          ADL_CMD_TYPE_ACT);
        // Subscribe to the 'at+mycmd' command again
        adl_atCmdSubscribe ("at+mycmd",
                           (adl_atCmdHandler_t)AT_MYCMD_Handler_2,
                          ADL_CMD_TYPE_ACT
    }
```

3.4 Timers

ADL supplies Timers Service interface to allow application tasks to require and handle timer related events.

The defined operations are:

- subscription functions (**adl_tmrSubscribe** & **adl_tmrSubscribeExt**) usable to require a timer event for the current task
- A **handler** call-back type (**adl_tmrHandler_t**) usable to receive timer related events
- An **unsubscription** function (**adl_tmrUnSubscribe**) usable to stop a currently running timer.

3.4.1 Required Header Files

The header file for the functions dealing with timers is:

adl_TimerHandler.h

3.4.2 The adl_tmr_t Structure

This structure is used to store timers related parameters. **adl_tmrSubscribe** and **adl_tmrSubscribeExt** return a pointer on this structure, which will be usable later to unsubscribe from the timer through **adl_tmrUnSubscribe**.

- Code:

```
typedef struct
{
      u8                      TimerId;
      adl_tmrCyclicMode_e     bCyclic;
      adl_tmrType_e           TimerType;
      u32                     TimerValue;
      adl_tmrHandler_t        TimerHandler;
} adl_tmr_t;
```

- Description

  TimerId

    0 based internal timer identifier. This identifier will be provided to **adl_tmrHandler_t handler** on each call.

  bCyclic

    Remembers the associated timer cyclic mode.

  TimerType

    Remembers the programmed timer granularity.

  TimerValue

    Remembers the programmed timer duration.

TimerHandler

Remembers the timer handler address, provided at subscription time.

3.4.3 Defines

3.4.3.1 ADL_TMR_MS_TO_TICK

Several conversion from timing unit to ticks.

- Code

```
#define ADL_TMR_MS_TO_TICK(MsT)     ((u32)(((MsT)*7)+64)>>7)
```

- Description

ADL_TMR_MS_TO_TICK(MsT): **Timer conversion from milliseconds to ticks**

3.4.3.2 ADL_TMR_100MS_TO_TICK

Several conversion from timing unit to ticks.

- Code

```
#define ADL_TMR_100MS_TO_TICK(MsT)  ((u32)(((MsT)*693L)+64)>>7)
```

- Description

ADL_TMR_100MS_TO_TICK(MsT): From 100 milliseconds to ticks

3.4.3.3 ADL_TMR_S_TO_TICK

Several conversion from timing unit to ticks.

- Code

```
#define ADL_TMR_S_TO_TICK(SecT)     ((u32)(((SecT)*6934L)+64)>>7)
```

- Description

ADL_TMR_S_TO_TICK(SecT): From seconds to ticks

3.4.3.4 ADL_TMR_MN_TO_TICK

Several conversion from timing unit to ticks.

- Code

```
#define ADL_TMR_MN_TO_TICK(MnT)     ((u32)(((MnT)*416034L)+64)>>7)
```

- Description

ADL_TMR_MN_TO_TICK(MnT): From minutes to ticks

3.4.4 The adl_tmrType_e

Allows to define the granularity (time unit) for the **adl_tmrSubscribe**, **adl_tmrSubscribeExt** & **adl_tmrUnSubscribe** functions.

- Code

```
typedef enum
{
        ADL_TMR_TYPE_100MS,
        ADL_TMR_TYPE_TICK,
        ADL_TMR_TYPE_LAST
} adl_audiotmrType_e;
```

- Description

ADL_TMR_TYPE_100MS:	100ms granularity timer.
ADL_TMR_TYPE_TICK:	18.5ms ticks granularity timer.
ADL_TMR_TYPE_LAST:	Reserved for internal use.

3.4.5 The adl_tmrCyclicMode_e

Allows to define the required cyclic option at timer subscription time.

Note:

When using the **ADL_TMR_CYCLIC_OPT_ON_EXPIRATION** option, there is no minimum time guaranteed between two timer events, since if the application is preempted for some time, timer events will continue to be generated even if the application is not notified.

This is not the case with the **ADL_TMR_CYCLIC_OPT_ON_RECEIVE** option: since the timer is re-programmed only when the application is notified, the duration between two events is guaranteed to be at least equal to the timer period.

- Code

```
typedef enum
{
        ADL_TMR_CYCLIC_OPT_NONE,
        ADL_TMR_CYCLIC_OPT_ON_EXPIRATION,
        ADL_TMR_CYCLIC_OPT_ON_RECEIVE,
        ADL_TMR_CYCLIC_OPT_LAST
} adl_tmrCyclicMode_e;
```

- Description

ADL_TMR_CYCLIC_OPT_NONE:	One shot timer: the timer will be automatically be unsubscribed as soon as the event is notified to the application.
ADL_TMR_CYCLIC_OPT_ON_EXPIRATION:	Cyclic timer, which will be re-programmed on expiration, just before the event is sent to the application.
ADL_TMR_CYCLIC_OPT_ON_RECEIVE:	Cyclic timer, which will be re-programmed on event reception, just before notifying the application's handler.
ADL_TMR_CYCLIC_OPT_LAST:	Reserved for internal use.

3.4.6 The adl_tmr_Handler_t

Call-back function, provided in an **adl_tmrSubscribe** or **adl_tmrSubscribeExt** call, and notified each time the related timer occurs.

- Prototype:

```
typedef void(*) adl_tmr_Handler_t (u8           ID,
                                   void *       Context );
```

- Parameters

ID

Timer internal identifier (readable from the **adl_tmr_t** pointer returned at subscription time).

Context

Pointer on the application context provided to **adl_tmrSubscribeExt** function. Will be set to NULL is the timer was programmed with **adl_tmrSubscribe** function.

Note:

Such a call-back function will always be called in the task context where the timer was programmed with **adl_tmrSubscribe** or **adl_tmrSubscribeExt**.

Timer events should be delayed if the applicative task is pre-empted due to higher priority (applicative or firmware) tasks processing.

3.4.7 The adl_tmrSubscribe Function

This function starts a timer with an associated callback function. The callback function will be executed as soon as the timer expires, in the task context where the **adl_tmrSubscribe** function was called.

- Prototype

```
adl_tmr_t *adl_tmrSubscribe( bool              bCyclic,
                             u32               TimerValue,
                             adl_tmrType_e     TimerType,
                             adl_tmrHandler_t  Timerhdl )
```

- Parameters

bCyclic:

This boolean flag indicates whether the timer is cyclic (TRUE) or not (FALSE). A cyclic timer is automatically restarted before calling the application event handler.

TimerValue:

The number of periods after which the timer expires (depends on TimerType parameter required time unit).

TimerType:

Unit of the TimerValue parameter (uses the **adl_tmrType_e** type).

Timerhdl:

The callback function associated to the timer (using the **adl_tmrHandler_t** type).

- Returned values
  - o A positive timer handle (an **adl_tmr_t** pointer) on success, usable to unsubscribe later from the timer service; a NULL or negative value (the timer is not started)
  - o NULL If TimerValue is 0, or if there is no more timer ressource for the current task. A task can use up to 32 timers at the same time.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low or high level interrupt handler (the function is forbidden in this context).

<u>Note:</u>

Since the Wireless CPU® time granularity is 18.5 ms, the 100 ms steps are emulated, reaching a value as close as possible to the requested one modulo 18.5. E.g., if a 20 * 100ms timer is required, the real time value will be 1998 ms (108 * 18.5ms).

The maximal value of "TimerValue" parameter is 0x5E9000 when "**ADL_TMR_TYPE_100MS**" timer is subscribed.

3.4.8 The adl_tmrSubscribeExt Function

This function starts a timer with an associated callback function. The callback function will be executed as soon as the timer expires, in the task context where the **adl_tmrSubscribe** function was called.

- Prototype

```
adl_tmr_t *adl_tmrSubscribeExt (adl_tmrCyclicMode_e   CyclicOpt,
                                u32                   TimerValue,
                                adl_tmrType_e         TimerType,
                                adl_tmrHandler_t      Timerhdl,
                                void *                Context);
```

- Parameters

CyclicOpt:

This option flag allows to set the required cyclic mode of the timer, using the **adl_tmrCyclicMode_e** type.

TimerValue:

The number of periods after which the timer expires (depends on TimerType parameter required time unit).

TimerType:

Unit of the TimerValue parameter (uses the **adl_tmrType_e** type).

Timerhdl:

The callback function associated to the timer (using the **adl_tmrHandler_t** type).

Context:

Pointer on an application defined context, which will be provided to the handler when the timer event will occur. This parameter should be set to NULL if not used.

- Returned values
  - o A positive timer handle (an **adl_tmr_t** pointer) on success, usable to unsubscribe later from the timer service; on error, a NULL or negative value (the timer is not started).
  - o NULL If TimerValue is 0 or too big, or if there is no more timer resource for the current task. A task can use up to 32 timers at the same time.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low or high level interrupt handler (the function is forbidden in this context).

Note:

Since the Wireless CPU® time granularity is 18.5 ms, the 100 ms steps are emulated, reaching a value as close as possible to the requested one modulo 18.5. E.g., if a 20 * 100ms timer is required, the real time value will be 1998 ms (108 * 18.5ms).

3.4.9 The adl_tmrUnSubscribe Function

This function stops the timer and unsubscribes to it and his handler. The call to this function is only meaningful to a cyclic timer or a timer that has not expired yet.

- Prototype

```
s32 adl_tmrUnSubscribe( adl_tmr_t*        t,
                        adl_tmrHandler_t  Timerhdl,
                        adl_tmrType_e     TimerType )
```

- Parameters

t:

Timer handle to be unsubscribed, previously returned by **adl_tmrSubscribe** or **adl_tmrSubscribeExt**.

Timerhdl:

The callback function associated to the timer. This parameter is only used to verify the coherence of **t** parameter. It has to be the timer handler used in the subscription procedure.

For example:

```
PhoneTaskTimerPtr = adl_tmrSubscribe (TRUE, 10, OneSecond,
                    ADL_TMR_TYPE_100MS, PhoneTaskTimer);
// Later ......
adl_tmrUnSubscribe (PhoneTaskTimerPtr, PhoneTaskTimer,
                    ADL_TMR_TYPE_100MS) ;
```

TimerType:

Time unit of the returned value, using the **adl_tmrType_e** enumeration.

- Returned values
  - On success, a positive value indicating the remaining time of the timer before it expires (time unit depends on the TimerType parameter value); On failure, a negative error value: **ADL_RET_ERR_BAD_HDL** if the provided timer handle is unknown
  - **ADL_RET_ERR_BAD_STATE** if the timer has already expired.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low or high level interrupt handler (the function is forbidden in this context).

Note:

When the **ADL_RET_ERR_BAD_STATE** error code is returned, the timer is correctly unsubscribed. This error code occurs when the function is called after the timer has elapsed at hardware level, but before the timer handler is notified.

Once a "one shot" (non cyclic) timer has expired and the handler is called, there is no need to unsubscribe from the Timer service: such a timer is automatically unsubscribed once elapsed.

3.4.10 Example

The code sample below illustrates a nominal use case of the ADL Timers Service public interface (error cases are not handled).

```
adl_tmr_t *tt, *tt2;
u16 timeout_period = 5;     // in 100 ms steps;

void Timer_Handler( u8 Id, void * Context )
{
    // We do not unsubscribe to the timer because it has 'naturally' expired
    adl_atSendResponse(ADL_AT_RSP, "\r\Timer timed out\r\n");
}

void Timer_Handler2( u8 Id, void * Context )
{
    // Unsubscribe from the timer resource
    adl_tmrUnSubscribe ( tt2, Timer_Handler2 );
}

// main function
void adl_main ( adl_InitType_e adlInitType )
{
    // We set up a one-shot timer
    tt = adl_tmrSubscribe ( FALSE,
                            timeout_period,
                            ADL_TMR_TYPE_100MS,
                            Timer_Handler );

    // We set up a cyclic timer
    tt2 = adl_tmrSubscribeExt ( ADL_TMR_CYCLIC_OPT_NONE,
                                timeout_period,
                                ADL_TMR_TYPE_100MS,
                                Timer_Handler2,
                                NULL );
}
```

3.5 Memory Service

The ADL Memory Service allows the applications to handle dynamic memory buffers, and get information about the platform's RAM mapping.

The defined operations are:

- get & release functions **adl_memGet** & **adl_memRelease** usable to manage dynamic memory buffers
- An information function **adl_memGetInfo** usable to retrieve information about the platform's RAM mapping

3.5.1 Required Header File

The header file for the memory functions is:

adl_memory.h

3.5.2 Data Structures

3.5.2.1 The adl_memInfo_t Structure

This structure contains several fields containing information about the platform's RAM mapping.

Note:

The RAM dedicated to the Open AT® application is divided in three areas (Call stack, Heap memory & Global variables). The **adl_memGetInfo** function returns these area current sizes.

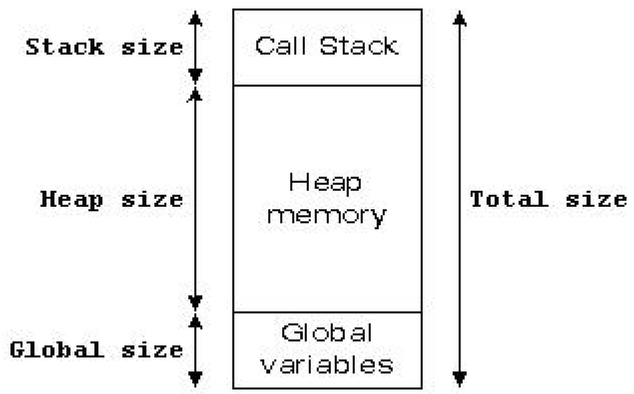


Figure 3: Open AT® RAM Mapping

- Code

```
typedef  structure
{
   u32            TotalSize
   u32            StackSize
   u32            HeapSize
   u32            GlobalSize
} adl_memInfo_t
```

- Description

TotalSize

Total RAM size for the Open AT® application (in bytes). Please refer to the 2.3 Memory Resources chapter for more information.

StackSize

Open AT® application call stacks area size (in bytes). This size is defined by the Open AT® application in the **adl_InitTasks** task table, and thanks to the **adl_InitIRQLowLevelStackSize** and **adl_InitIRQHighLevelStackSize** constants. (Please refer to the 3 Mandatory API chapter for more information.

Note:

This field is set to 0 under Remote Task Environment

HeapSize

Open AT® application total heap memory area size (in bytes). This size is the difference between the total Open AT® memory size and the Global & Stack areas sizes.

Note:

This field is set to 0 under Remote Task Environment

GlobalSize

Open AT® application global variables area size (in bytes). This size is defined at the binary link step; it includes the ADL library, plug-in libraries (if any) and Open AT® application global variables.

Note:

This field is set to 0 under Remote Task Environment

3.5.3 Defines

3.5.3.1 The adl_memRelease

This macro releases the allocated memory buffer designed by the supplied pointer.

- Parameters

  _p_

  A pointer on the allocated memory buffer

- Returned values

  - TRUE If the memory was correctly released. In this case, the provided pointer is set to NULL.

  Note:

  If the memory release fails, one of the following exceptions is generated (these exception cannot be filtered by the Error service, and systematically lead to a reset of the Wireless CPU®).

- Exceptions

  - RTK exception 155

    The supplied address is out of the heap memory address range

  - RTK exception 161 or 166

    The supplied buffer header or footer data is corrupted: a write overflow has occurred on this block

  - RTK exception 159 or 172

    The heap memory release process has failed due to a global memory corruption in the heap area.

3.5.4 The adl_memGetInfo Function

This function returns information about the Open AT® RAM areas sizes.

- Prototype

```
s32 adl_memGetInfo ( adl_memInfo_t * Info );
```

- Parameters

  Info:

  Please refer to the 3.5.2.1 **adl_memInfo_t** structure.

  - **TotalSize**

    Total RAM size for the Open AT® application (in bytes).
    Please refer to the 2.3 Memory Resources chapter for more information.

  - **StackSize**

    Open AT® application call stack area size (in bytes).
    This size is defined by the Open AT® application through the **wm_apmCustomStackSize** constant (Please refer to the 3 Mandatory API chapter for more information).

Note:

This field is set to 0 under Remote Task Environment.

- **HeapSize**

  Open AT® application total heap memory area size (in bytes).
  This size is the difference between the total Open AT® memory size and the Global & Stack areas sizes.

Note:

This field is set to 0 under Remote Task Environment.

- **GlobalSize**

  Open AT® application global variables area size (in bytes).
  This size is defined at the binary link step; it includes the ADL library, plug-in libraries (if any) and Open AT® application global variables.

Note:

This field is set to 0 under Remote Task Environment.

- Reminder:

  The Open AT® RAM is divided in three areas (Call stack, Heap memory & Global variables). This function returns the area sizes. Please refer to the Figure 3 Open AT® on RAM Mapping.

- Returned values
  - **OK** on success; the **Info** parameter is updated in the Open AT® RAM information.
  - **ADL_RET_ERR_PARAM** on parameter error

3.5.5 The adl_memGet Function

This function allocates the memory for the requested **size** into the client application RAM memory.

- Prototype

```
void * adl_memGet ( u32 size );
```

- Parameters

  size:

  The memory buffer requested size (in bytes).

- Returned values
  - A pointer to the allocated memory buffer on success.
- Exceptions
  - **ADL_ERR_MEM_GET** If the memory allocation fails, this function will lead to a **ADL_ERR_MEM_GET** error, which can be handled by the Error Service. If this error is filtered and refused by the error handler, the function will return NULL. Please refer to the paragraph 3.12 on Error service for more information.

- o **RTK exception 166**
  A buffer header or footer data is corrupted: a write overflow has occurred on this block.

Note:

Memory allocation may also fail due to an unrecoverable corrupted memory state; one of the following exceptions is then generated (these exceptions cannot be filtered by the Error service, and systematically lead to a reset of the Wireless CPU®).

3.5.6 The adl_memRelease Function

Internal memory release function, which should not be called directly. The **adl_memRelease** macro has to be used in order to release memory buffer.

- Prototype

  bool adl_memRelease (void ** ptr);

- Parameters

  ptr:

  A pointer on the allocated memory buffer.

- Returned values
  - Please refer to the 3.5.3.1 **adl_memRelease** macro definition.

3.5.7 Heap Memory Block Status

A list of the currently reserved heap memory blocks can be displayed at any time using the Target Monitoring Tool "Get RTK Status" command. Please refer to the Tools Manual (document [2]) for more information.

3.5.8 Example

This example demonstrates how to use the Memory service in a nominal case (error cases are not handled).

```
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    adl_memInfo_t MemInfo;
    u8 * MyByteBuffer

    // Gets Open AT® RAM information
    adl_memGetInfo ( &MemInfo );

    // Allocates a 10 bytes memory buffer
    MyByteBuffer = ( u8 * ) adl_memGet ( 10 );

    // Releases the previously allocated memory buffer
    adl_memRelease ( MyByteBuffer );
}
```

3.6 ADL Registry Service

The ADL Registry Service allows to give to Open AT® applications an access to the platform registry, used to store generic information about the software & hardware capabilities or configuration.

The defined operations are:

- An **adl_regGetWCPUType** function to retrieve information from the registry about current Wireless CPU® identifier
- An **adl_regGetHWInteger** function to retrieve integer value of a registry entry
- An **adl_regGetHWData** function to retrieve the data value of a registry entry
- An **adl_regGetHWDataChunk** function to retrieve the data value of a registry entry

3.6.1 Required Header File

The header file is:

```
adl_reg.h
```

3.6.2 The adl_regGetWCPUType Function

This function allows the application to retrieve the current Wireless CPU® identifier

- Prototype

```
s32 adl_regGetWCPUType  ( ascii *  CPUType   );
```

- Parameters

CPUType:

String buffer where the Wireless CPU® type identifier has to be copied.

Can be set to NULL in order just to retrieve the required string buffer size.

- Returned values

Positive number of copied characters to the supplied string buffer (including terminal 0).

3.6.3 The adl_regGetHWInteger Function

This function allows the application to retrieve the integer value of a registry entry.

- Prototype

```
s32 adl_regGetHWInteger  ( ascii *       Label,
                           s32 *         Value   );
```

- Parameters

Label

Label of the entry in the registry.

Value

Integer buffer where the value of the registry label has to be copied.

- Returned values
  - A **OK** on success.
  - A negative error value otherwise:
    - **ADL_RET_UNKNOWN_HDL** if the registry Label is not found.
    - **ADL_RET_BAD_HDL** if the registry type required is not good.
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value

3.6.4 The adl_regGetHWData Function

This function allows the application to retrieve the data value of a registry entry.

- Prototype

```
s32 adl_regGetHWData  ( ascii *    Label,
                        void *     Data   );
```

- Parameters

Label

Label of the entry in the registry.

Data

Data buffer where the information of the registry label has to be copied,

This is an optional parameter and must be set to 0 if not used.

- Returned values
  - The size of the Data information on success.
  - A negative error value otherwise:
    - **ADL_RET_UNKNOWN_HDL** if the registry Label is not found.
    - **ADL_RET_BAD_HDL** if the registry type required is not good.
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.

3.6.5 The adl_regGetHWDataChunk Function

This function allows the application to retrieve the data value of a registry entry.

- Prototype

```
s32 adl_regGetHWDataChunk  ( ascii *    Label,
                             void *     Data
                             u32        BeginOffset,
                             u32        ByteCount   );
```

- Parameters

  Label

  Label of the entry in the registry.

  Data

  Data buffer where the information of the registry label has to be copied.

  This is an optional parameter and must be set to 0 if not used.

  BeginOffset

  Offset within the data value, this is an optional parameter must be set to 0 if not used

  ByteCount

  Number of bytes to get, this is an optional parameter must be set to 0 if not used

- Returned values
  - The size of the Data information on success.
  - A negative error value otherwise:
    - **ADL_RET_UNKNOWN_HDL** if the registry Label is not found.
    - **ADL_RET_BAD_HDL** if the registry type required is not good.
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.

3.6.6 Example

```
// Retrieve Wireless CPU® identifier
  void * function_1()
  {
      // Retrieve required size for Wireless CPU® identifier
      u32 NameSize = adl_regGetWCPUType ( NULL );

      // Allows enough memory
      ascii * Name = adl_memGet ( NameSize );

      // Retrieve Wireless CPU® type
      adl_regGetWCPUType ( Name );

      // Check current Wireless CPU® type
      if ( !wm_strcmp ( Name, "WMP100" ) )
      {
        // WMP100 Wireless CPU®
      }
      else if ( !wm_strcmp ( Name, "Q2686" ) )
      {
        // Q2686 Wireless CPU®
      }
      else if ( !wm_strcmp ( Name, "Q2687" ) )
      {
        // Q2687 Wireless CPU®
      }
```

```
}

// Retrieve hardware integer information
void * function_2()
{
    u32 Hardware_info;

    // Retrieve the integer information
    adl_regGetHWInteger ( "Hardware_info_label", &Hardware_info );
    ...
}

// Retrieve hardware data information
void * function_3()
{
    // Retrieve required size for hardware data information
    u32 Hardware_info_size = adl_regGetHWData ( "Hardware_info_label",
                                                NULL );

    // Allows enough memory
    adl_HardwareInfoExample_t * Hardware_info_data = adl_memGet
                                                  ( Hardware_info_size );

    // Retrieve the adl_HardwareInfoExample_t information
    adl_regGetHWData ( "Hardware_info_label", Hardware_info_data );
    ...
}

// Retrieve hardware data information
void * function_4()
{
    // Allows enough memory for a part of hardware data information
    ascii * Hardware_info_data_chunk = adl_memGet ( 10 );

    // Retrieve the adl_HardwareInfoExample_t information
    adl_regGetHWDataChunk ( "Hardware_info_label",
                          Hardware_info_data_chunk , 5 , 10 );
    ...
}
```

3.7 Debug Traces

This service allows to display debug trace strings on the Target Monitoring Tool. The different ways to embed these trace strings in an Open AT® application depends on the selected configuration in the used Open AT® IDE (see below).

For more information about the Target Monitoring Tool, the configurations and the Integrated Development Environment, please refer to the Tools Manual (document [2]).

The defined operations are:

- Trace function & macros (**adl_trcPrint**, **TRACE** & **FULL_TRACE**) to print the required trace string
- Dump function & macros (**adl_trcDump**, **DUMP** & **FULL_DUMP**) to dump the required buffer content

3.7.1 Required Header File

The header file for the flash functions is:

adl_traces.h

3.7.2 Build Configuration Macros

According to the chosen build configuration in the IDE, following macros will be defined or not, allowing the user to embed none, part or the entire debug traces information in its final application.

3.7.2.1 Debug Configuration

When the Debug configuration is selected in the IDE, the __DEBUG_APP__ compilation flag is defined, and also the **TRACE & DUMP** macros.

Traces & dumps declared with these macros will be embedded at compilation time.

In this Debug configuration, the **FULL_TRACE** and **FULL_DUMP** macros are ignored (even if these are used in the application source code, they will neither be compiled nor displayed on Target Monitoring Tool at runtime).

3.7.2.2 Full Debug Configuration

When the Full Debug configuration is selected in the IDE, both the __DEBUG_APP__ and __DEBUG_FULL__ compilation flags are defined, and also the **TRACE, FULL_TRACE, DUMP** & **FULL_DUMP** macros.

Traces & dumps declared with these macros will be embedded at compilation time.

3.7.2.3 Release Configuration

When the Release configuration is selected in the IDE, neither the __DEBUG_APP__ nor __DEBUG_FULL__ compilation flags are defined

The **TRACE, FULL_TRACE, DUMP** and **FULL_DUMP** macros are ignored (even if these ones are used in the application source code, they will neither be compiled, nor displayed on Target Monitoring Tool at runtime).

3.7.2.4 Defines

3.7.2.4.1 TRACE

This macro is a shortcut to the **adl_trcPrint** function. Traces declared with this macro are only embedded in the application if it is compiled with in the Debug or Full Debug configuration, but not in the Release configuration.

```
#define TRACE  ( _X_    )
```

3.7.2.4.2 DUMP

This macro is a shortcut to the **adl_trcDump** function. Dumps declared with this macro are only embedded in the application if it is compiled with in the Debug or Full Debug configuration, but not in the Release configuration.

```
#define DUMP  ( _lvl_,
                _P_,
                _L_    )
```

3.7.2.4.3 FULL TRACE

This macro is a shortcut to the adl_trcPrint function. Traces declared with this macro are only embedded in the application if it is compiled with in Full Debug configuration, but not in the Debug or Release configuration.

```
#define FULL_TRACE  ( _X_    )
```

3.7.2.4.4 FULL DUMP:

This macro is a shortcut to the **adl_trcDump** function. Dumps declared with this macro are only embedded in the application if it is compiled with in Full Debug configuration, but not in the Debug or Release configuration.

```
#define FULL_DUMP  ( _lvl_,
                     _P_,
                     _L_    )
```

3.7.3 The adl_trcPrint Function

This function displays the required debug trace on the provided trace level. The trace will be displayed in the Target Monitoring Tool, according to the current context:

- for tasks: on the trace element name defined in the tasks declaration table (cf. Application Initialization service)
- for Low Level Interrupt handlers: on the "LLH" trace element
- for High Level Interrupt handlers: on the "HLH" trace element

In addition to the trace information, a Wireless CPU® local timestamp is also displayed in the tool.

Example1:

```
u8 I = 123;
```

```
TRACE (( 1, "Value of I: %d", I ));
```

At runtime, this will display the following string on the CUS4 level 1 on the Target Monitoring Tool:

```
Value of I: 123
```

- Prototype

```
s8 adl_trcPrint ( u8                  Level,
                  const ascii*        strFormat,
                                      ...  );
```

- Parameters

Level:

Trace level on which the information has to be sent. Valid range is **1 - 32**.

strFormat:

String to be displayed, using a standard C "sprintf" format.

...:

Additional arguments to be dynamically inserted in the provided constant string.

Note:

- Direct use of the **adl_trcPrint** function is not recommended. The **TRACE** & **FULL_TRACE** macros should be used instead, to take benefit of the build configurations features.
- '%s' character, normally used to insert strings, is not supported by the trace function.
- The trace display should be limited to 255 bytes. If the trace string is longer, it will be truncated.

3.7.4 The adl_trcDump Function

This function dumps the required buffer content on the provided trace level. The dump will be displayed in the Target Monitoring Tool, according to the current context:

- for tasks: on the trace element name defined in the tasks declaration table (cf. Application Initialization service)
- for Low Level Interrupt handlers: on the "LLH" trace element
- for High Level Interrupt handlers: on the "HLH" trace element

In addition to the trace information, a Wireless CPU® local timestamp is also displayed in the tool.

Since a display line maximum length is 255 bytes, if the display length is greater than 80 (each byte is displayed on 3 ascii characters), the dump will be segmented on several lines. Each 80 bytes truncated line will end with the "..." characters sequence.

Example 1

```
u8 * Buffer = "\x0\x1\x2\x3\x4\x5\x6\x7\x8\x9";
DUMP ( 1, Buffer, 10 );
```

At runtime, this will display the following string on the level 1 in the Target Monitoring Tool:

```
00 01 02 03 04 05 06 07 08 09
```

Example 2

```
u8 Buffer [ 200 ], i;
for ( i = 0 ; i < 200 ; i++ ) Buffer [ i ] = i;
DUMP ( 1, Buffer, 200 );
```

At runtime, this will display the following three lines on the level 1 in the Target Monitoring Tool:

```
00 01 02 03 04 05 06 07 08 09 0A [bytes from 0B to 4D] 4E 4F...
50 51 52 53 54 55 56 57 58 59 5A [bytes from 5B to 9D] 9E 9F...
A0 A1 A2 A3 A4 A5 A6 A7 [bytes from A8 to C4] C5 C6 C7
```

- Prototype

```
void adl_trcDump ( u8              Level,
                   u8 *            DumpBuffer,
                   u16             DumpLength );
```

- Parameters

Level:

Trace level on which the information has to be sent. Valid range is **1 - 32**.

DumpBuffer:

Buffer address to be dumped.

DumpLength:

Number of bytes to be displayed at required address.

<u>Note:</u>

Direct use of the **adl_trcDump** function is not recommended. The **DUMP** & **FULL_DUMP** macros should be used instead, to take benefit of the build configurations features.

3.7.5 Example

The code sample below illustrates a nominal use case of the ADL Debug Traces service public interface (error cases are not handled).

```
u8 MyInt = 12;
ascii * MyString = "hello";

// Print a debug trace for current context on level 1
TRACE (( 1, "My Sample Trace: %d", MyInt ));

// Dump a buffer content for current context on level 2
DUMP ( 2, MyString, strlen ( MyString ) );
```

3.8 Flash

3.8.1 Required Header File

The header file for the flash functions is:

```
adl_flash.h
```

3.8.2 Flash Objects Management

An ADL application may subscribe to a set of objects identified by an handle, used by all ADL flash functions.

This handle is chosen and given by the application at subscription time.

To access to a particular object, the application gives the handle and the ID of the object to access.

At first subscription, the Handle and the associated set of IDs are saved in flash. The number of flash object IDs associated to a given handle may be only changed after have erased the flash objects (with the AT+WOPEN=3 command).

For a particular handle, the flash objects ID take any value, from 0 to the ID range upper limit provided on subscription.

Important note:

Due to the internal storage implementation, only up to 2000 object identifiers can exist at the same time.

3.8.2.1 Flash objects write/erase inner process overview

Written flash objects are queued in the flash object storage place. Each time the **adl_flhWrite** function is called, the process below is done:

- If the object already exists, it is now considered as "erased" (ie. "adl_flhWrite(X);" <=> "adl_flhDelete(X); adl_flhWrite(X);")
- The flash object driver checks if there is enough place the store the new object. If not, a Garbage Collector process is done (see below).
- The new object is created.

About the erase process, each time the **adl_flhDelete** (or **adl_flhWrite**) function is called on a ID, this object is from this time "considered as erased", even if it is not physicaly erased (an inner "erase flag" is set on this object).

Objects are physically erased only when the Garbage Collector process is done, when an **adl_flhWrite** function call needs a size bigger than the available place in the flash objects storage place. The Garbage Collector process erases the flash objects storage place, and re-write only the objects which have not their "erase flag" set.

Please note that the flash memory physical limitation is the erasure cycle number, which is granted to be at least 100.000 times.

Caution:

The Garbage Collector process is a time consuming operation. Performing numerous flash write operations in the same event handler increases the probability of Garbage Collector occurence, and should lead to a watchdog reset of the Wireless CPU®. It is not recommended to perform too many flash write operations in the same event handler. If numerous operations are required, it is advised to regularly "give back the hand" to the Firmware (by introducing timers) in the write loop, in order to avoid the Watchdog reset to occur.

3.8.2.2 Flash Objects in Remote Task Environment

When an application is running in Remote Task Environment, the flash object storage place is emulated on the PC side: objects are read/written from/to files on the PC hard disk, and not from/to the Wireless CPU®s flash memory. The two storage places (Wireless CPU® and PC one) may be synchronized using the RTE Monitor interface (cf. the Tools Manual [2] for more information).

3.8.3 The adl_flhSubscribe Function

This function subscribes to a set of objects identified by the given Handle.

- Prototype

```
s8 adl_flhSubscribe ( ascii* Handle, u16 NbObjectsRes)
```

- Parameters

Handle:

The Handle of the set of objects to subscribe to.

NbObjectRes :

The number of objects related to the given handle. It means that the IDs available for this handle are in the range [0 , (NbObjectRes – 1)].

- Returned values
  - **OK** on success (first allocation for this handle)
  - **ADL_RET_ERR_PARAM** on parameter error,
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if space is already created for this handle,
  - **ADL_FLH_RET_ERR_NO_ENOUGH_IDS** if there are no more enough object IDs to allocate the handle.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

Note:

Only one subscription is necessary. It is not necessary to subscribe to the same handle at each application start.

It is not possible to unsubscribe from an handle. To release the handle and the associated objects, the user must do an AT+WOPEN=3 to erase the flash objects of the Open AT® Embedded Application.

3.8.4 The adl_flhExist Function

This function checks if a flash object exists from the given Handle at the given ID in the flash memory allocated to the ADL developer.

- Prototype

  `s32 adl_flhExist (ascii* Handle, u16 ID )`

- Parameters

  Handle:

  The Handle of the subscribe set of objects.

  ID:

  The ID of the flash object to investigate (in the range allocated to the provided Handle).

- Returned values
  - the requested Flash object length on success
  - **0K** if the object does not exist.
  - **ADL_RET_ERR_UNKNOWN_HDL** if handle is not subscribed
  - **ADL_FLH_RET_ERR_ID_OUT_OF_RANGE** if ID is out of handle range
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.8.5 The adl_flhErase Function

This function erases the flash object from the given Handle at the given ID.

- Prototype

  `s8 adl_flhErase (ascii* Handle, u16 ID )`

- Parameters

  Handle:

  The Handle of the subscribed set of objects.

  ID:

  The ID of the flash object to be erased.

  Important note:

  If ID is set to **ADL_FLH_ALL_IDS**, all flash objects related to the provided handle will be erased.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_UNKNOWN_HDL** if handle is not subscribed
  - **ADL_FLH_RET_ERR_ID_OUT_OF_RANGE** if ID is out of handle range
  - **ADL_FLH_RET_ERR_OBJ_NOT_EXIST** if the object does not exist
  - **ADL_RET_ERR_FATAL** if a fatal error occurred (**ADL_ERR_FLH_DELETE** error event will then be generated)

- o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.8.6 The adl_flhWrite Function

This function writes the flash object from the given Handle at the given ID, for the length provided with the buffer provided. A single flash object can use up to 30 Kbytes of memory.

- Prototype

```
s8 adl_flhWrite (ascii* Handle, u16 ID, u16 Len, u8 *WriteData )
```

- Parameters

Handle:

The Handle of the subscribed set of objects.

ID:

The ID of the flash object to write.

Len:

The length of the flash object to write.

WriteData:

The provided string to write in the flash object.

- Returned values
  - o **OK** on success
  - o **ADL_RET_ERR_PARAM** if one at least of the parameters has a bad value.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if handle is not subscribed
  - o **ADL_FLH_RET_ERR_ID_OUT_OF_RANGE** if ID is out of handle range
  - o **ADL_RET_ERR_FATAL** if a fatal error occurred (ADL_ERR_FLH_WRITE error event will then occur).
  - o **ADL_FLH_RET_ERR_MEM_FULL** if flash memory is full.
  - o **ADL_FLH_RET_ERR_NO_ENOUGH_IDS** if the object can not be created due to the global ID number limitation.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.8.7 The adl_flhRead Function

This function reads the flash object from the given Handle at the given ID, for the length provided and stores it in a buffer.

- Prototype

```
s8 adl_flhRead (ascii* Handle, u16 ID, u16 Len, u8 *ReadData )
```

- Parameters

  Handle:

  The Handle of the subscribed set of objects

  ID:

  The ID of the flash object to read.

  Len:

  The length of the flash object to read.

  ReadData:

  The string allocated to store the read flash object.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** if one at least of the parameters has a bad value.
  - **ADL_RET_ERR_UNKNOWN_HDL** if handle is not subscribed
  - **ADL_FLH_RET_ERR_ID_OUT_OF_RANGE** if ID is out of handle range
  - **ADL_FLH_RET_ERR_OBJ_NOT_EXIST** if the object does not exist.
  - **ADL_RET_ERR_FATAL** if a fatal error occurred (ADL_ERR_FLH_READ error event will then occur).
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.8.8 The adl_flhGetFreeMem Function

This function gets the current remaining flash memory size.

- Prototype

```
u32 adl_flhGetFreeMem ( void )
```

- Returned values
  - Current free flash memory size in bytes.

3.8.9 The adl_flhGetIDCount Function

This function returns the ID count for the provided handle, or the total remaining ID count.

- Prototype

```
s32 adl_flhGetIDCount (ascii* Handle)
```

- Parameters

  Handle:

  The Handle of the subscribed set of objects. If set to NULL, the total remaining ID count will be returned.

* Returned values
  * On success:
    * ID count allocated on the provided handle if any;
    * the total remaining ID count if the handle is set to NULL
  * **ADL_RET_ERR_UNKNOWN_HDL** if handle is not subscribed
  * **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.8.10 The adl_flhGetUsedSize Function

This function returns the used size by the provided ID range from the provided handle. The handle should also be set to NULL to get the whole used size.

* Prototype

```
s32 adl_flhGetUsedSize (ascii*      Handle,
                        u16         StartID,
                        u16         EndID)
```

* Parameters

Handle:

The Handle of the subscribed set of objects. If set to NULL, the whole flash memory used size will be returned.

StartID:

First ID of the range from which to get the used size ; has to be lower than EndID.

EndID:

Last ID of the range from which to get the used size; has to be greater than StartID. To get the used size by all an handle IDs, the [0 , ADL_FLH_ALL_IDS] range may be used

* Returned values
  * Used size on success: from the provided Handle if any, otherwise the whole flash memory used size
  * **ADL_RET_ERR_PARAM** on parameter error
  * **ADL_RET_ERR_UNKNOWN_HDL** if handle is not subscribed
  * **ADL_FLH_RET_ERR_ID_OUT_OF_RANGE** if ID is out of handle range
  * **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.9 FCM Service

ADL provides a FCM (Flow Control Manager) service to handle all FCM events, and to access to the data ports provided on the product.

An ADL application may subscribe to a specific flow (UART 1, UART 2 or USB physical/virtual ports, GSM CSD call data port, GPRS session data port or Bluetooth virtual data ports) to exchange data on it.

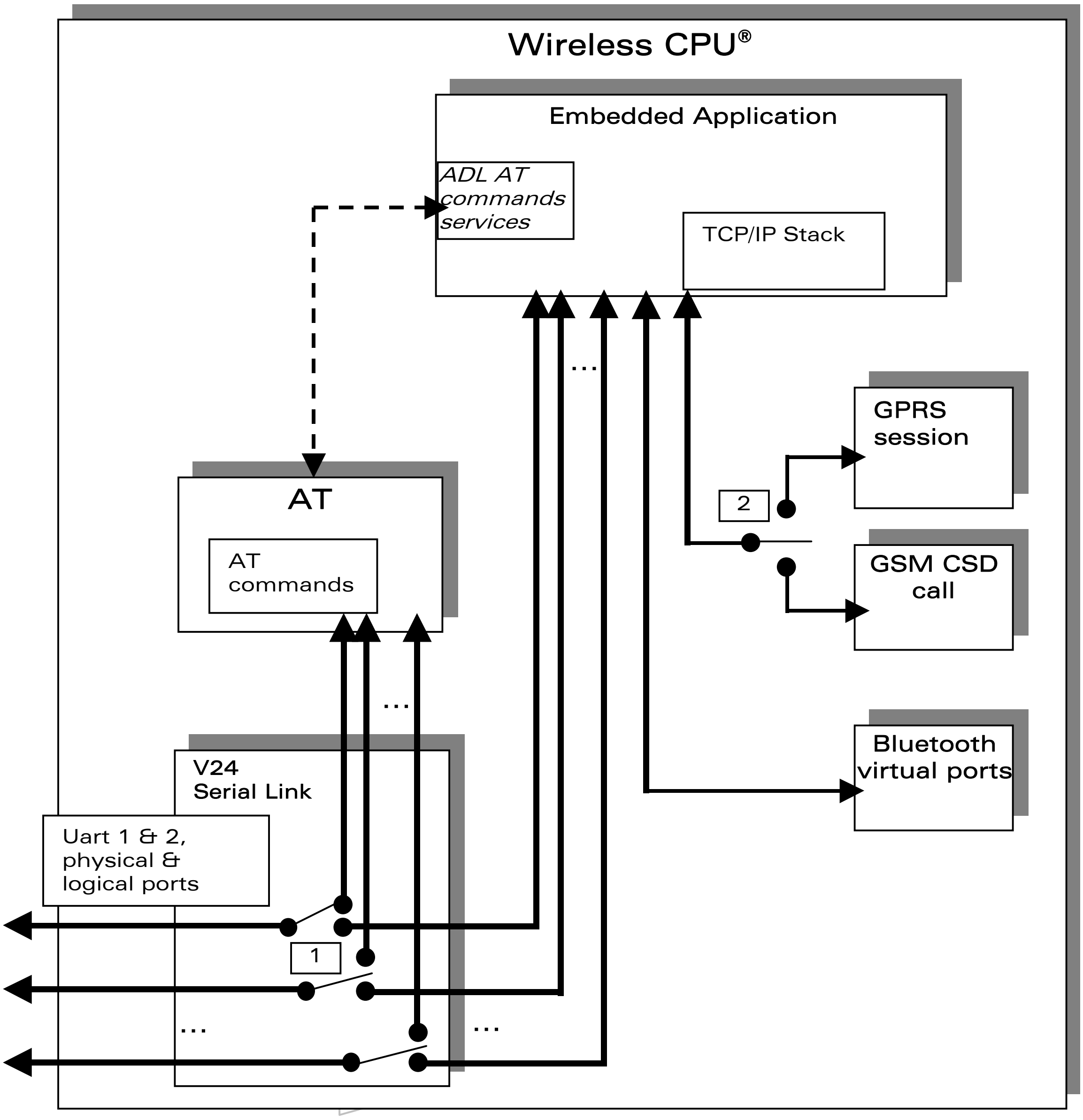


Figure 4: Flow Control Manager Representation

By default (ie. without any Open AT® application, or if the application does not use the FCM service), all the Wireless CPU®s ports are processed by the Wavecom Firmware. The default behaviors are:

- When a GSM CSD call is set up, the GSM CSD data port is directly connected to the UART port where the ATD command was sent ;
- When a GPRS session is set up, the GPRS data port is directly connected to the UART port where the ATD or AT+CGDATA command was sent ;
- When a Bluetooth peripheral is detected & connected through an SPP based profile, a local data bridge may be set up between a Bluetooth virtual data port and the required UART port, using the AT+WLDB command.

Once subscribed by an Open AT® application with the FCM service, a port is no more available to be used with the AT commands by an external application. The available ports are the ones listed in the ADL AT/FCM Ports service:

- ADL_PORT_UART_X / ADL_PORT_UART_X_VIRTUAL_BASE identifiers may be used to access to the Wireless CPU®s physicals UARTS, or logical 27.010 protocol ports ;
- ADL_PORT_GSM_BASE identifier may be used to access to a remote modem (connected through a GSM CSD call) data flow ;
- ADL_PORT_GPRS_BASE identifier may be used to exchange IP packets with the operator network and the Internet ;
- ADL_PORT_BLUETOOTH_VIRTUAL_BASE may be used to access to a connected Bluetooth device data stream with the Serial Port Profile (SPP).

The "1" switchs on the figure above means that UART based ports may be used with AT commands or FCM services as well. These switches are processed by the adl_fcmSwitchV24State function.

The "2" switch on the figure above means that either the GSM CSD port or the GPRS port may be subscribed at one time, but not both together.

Important note:

GPRS provides only **packet** mode transmission. This means that the embedded application can only send/receive **IP packets** to/from the GPRS flow.

3.9.1 Required Header File

The header file for the FCM functions is:

```
adl_fcm.h
```

3.9.2 The adl_fcmIsAvailable Function

This function allows to check if the required port is available and ready to handle the FCM service.

- Prototype

```
bool   adl_fcmIsAvailable     (adl_fcmFlow_e   Flow );
```

- Parameters

  Flow:

  Port from which to require the state.

- Returned values
  - **TRUE** if the port is ready to handle the FCM service
  - **FALSE** if it is not ready

  Notes:

  All ports should be available for the FCM service, except:
  - The Open AT® virtual one, which is only usable for AT commands,
  - The Bluetooth virtual ones with enabled profiles other than the SPP one,
  - If the port is already used to handle a feature required by an external application through the AT commands (+WLDB command, or a CSD/GPRS data session is already running)

3.9.3 The adl_fcmSubscribe Function

This function subscribes to the FCM service, opening the requested port and setting the control and data handlers. The subscription will be effective only when the control event handler has received the **ADL_FCM_EVENT_FLOW_OPENNED** event.

Each port may be subscribed only one time.

Additional subscriptions may be done, using the **ADL_FCM_FLOW_SLAVE** flag (see below). Slave subscribed handles will be able to send & receive data on/from the flow, but will know some limitations:

- For serial-line flows (UART physical & logical based ports), only the main handle will be able to switch the Serial Link state between AT & Data mode;
- If the main handle unsubscribe from the flow, all slave handles will also be unsubscribed.

Important note:

For serial-link related flows (UART physical & logical based ports), the corresponding port has to be opened first with the AT+WMFM command (for physical ports), or with the 27.010 protocol driver on the external application side (for logical ports), otherwise the subscription will fail. See AT Commands Interface guide (document [1]) for more information.

By default, only the UART1 physical port is opened.

A specific port state may be known using the ADL AT/FCM port service.

- Prototype

```
s8    adl_fcmSubscribe (     adl_fcmFlow_e     Flow,
                             adl_fcmCtrlHdlr_f CtrlHandler,
                             adl_fcmDataHdlr_f DataHandler );
```

- Parameters

Flow:

The allowed values are the available ports of the **adl_port_e** type. Only ports with the FCM capability may be used with this service (ie. all ports except the **ADL_PORT_OPEN_AT_VIRTUAL_BASE** and not SPP **ADL_PORT_BLUETOOTH_VIRTUAL_BASE** based ones).

Please note that the **adl_fcmFlow_e** type is the same than the **adl_port_e** one, except the fact that is may handle some additional FCM specific flags (see below). Previous versions FCM flows identifiers have been kept for ascendant compatibility. However, these constants should be considered as deprecated, and the **adl_port_e** type members should now be used instead.

```
#define ADL_FCM_FLOW_V24_UART1   ADL_PORT_UART1
#define ADL_FCM_FLOW_V24_UART2   ADL_PORT_UART2
#define ADL_FCM_FLOW_V24_USB     ADL_PORT_USB
#define ADL_FCM_FLOW_GSM_DATA    ADL_PORT_GSM_BASE
#define ADL_FCM_FLOW_GPRS        ADL_PORT_GPRS_BASE
```

To perform a slave subscription (see above), a bit-wise or has to be done with the flow ID and the **ADL_FCM_FLOW_SLAVE** flag ; for example:

```
adl_fcmSubscribe (ADL_PORT_UART1 | ADL_FCM_FLOW_SLAVE,
                  MyCtrlHandler,   MyDataHandler );
```

CtrlHandler:

FCM control events handler, using the following type:

```
typedef bool ( * adl_fcmCtrlHdlr_f ) (adl_fcmEvent_e event );
```

The FCM control events are defined below (All handlers related to the concerned flow (master and slaves) will be notified together with these events):

- **ADL_FCM_EVENT_FLOW_OPENNED** (related to **adl_fcmSubscribe**),
- **ADL_FCM_EVENT_FLOW_CLOSED** (related to **adl_fcmUnsubscribe**),
- **ADL_FCM_EVENT_V24_DATA_MODE** (related to **adl_fcmSwitchV24State**),
- **ADL_FCM_EVENT_V24_DATA_MODE_EXT** (see note below),
- **ADL_FCM_EVENT_V24_AT_MODE** (related to **adl_fcmSwitchV24State**),
- **ADL_FCM_EVENT_V24_AT_MODE_EXT** (see note below),
- **ADL_FCM_EVENT_RESUME** (related to **adl_fcmSendData** and **adl_fcmSendDataExt**),
- **ADL_FCM_EVENT_MEM_RELEASE** (related to **adl_fcmSendData** and **adl_fcmSendDataExt**) ,

This handler return value is not relevant, except for **ADL_FCM_EVENT_V24_AT_MODE_EXT**.

DataHandler:

FCM data events handler, using the following type:

```
typedef bool ( * adl_fcmDataHdlr_f ) ( u16 DataLen, u8 * Data );
```

This handler receives data blocks from the associated flow.

Once the data block is processed, the handler must return TRUE to release the credit, or FALSE if the credit must not be released. In this case, all credits will be released next time the handler will return TRUE.

On all flows, all data handlers (master and slaves) subscribed are notified with a data event, and the credit will be released only if all handlers return TRUE: each handler should return TRUE as default value.

If a credit is not released on the data block reception, it will be released the next time the data handler will return TRUE. The **adl_fcmReleaseCredits** should also be used to release credits outside of the data handler.

Maximum size of each data packets to be received by the data handlers depends on the flow type:

- On serial link flows (UART physical & logical based ports) : 120 bytes ;
- On GSM CSD data port : 270 bytes ;
- On GPRS port : 1500 bytes ;
- On Bluetooth virtual ports : 120 bytes.

If data size to be received by the Open AT® application exceeds this maximum packet size, data will be segmented by the Flow Control Manager, which will call several times the Data Handlers with the segmented packets.

Please note that on GPRS flow, whole IP packets will always be received by the Open AT® application.

- Returned values
  - A positive or null handle on success (which will have to be used in all further FCM operations). The Control handler will also receive a **ADL_FCM_EVENT_FLOW_OPENNED** event when flow is ready to process,
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value,
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the flow is already subscribed in master mode,
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if a slave subscription is made when master flow is not subscribed,
  - **ADL_FCM_RET_ERROR_GSM_GPRS_ALREADY_OPENNED** if a GSM or GPRS subscription is made when the other one is already subscribed.
  - **ADL_RET_ERR_BAD_STATE** if the required port is not available.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

Notes:

- When « 7 bits » mode is enabled on a v24 serial link, in data mode, payload data is located on the 7 least significant bits (LSB) of every byte.
- When a serial link is in data mode, if the external application sends the sequence "1s delay ; +++ ; 1s delay", this serial link is switched to AT mode, and corresponding handler is notified by the **ADL_FCM_EVENT_V24_AT_MODE_EXT** event. Then the behavior depends on the returned value.

  If it is TRUE, all this flow remaining handlers are also notified with this event. The main handle can not be un-subscribed in this state.

  If it is FALSE, this flow remaining handlers are not notified with this event, and this serial link is switched back immediately to data mode.

  In the first case, after the **ADL_FCM_EVENT_V24_AT_MODE_EXT** event, the main handle subscriber should switch the serial link to data mode with the **adl_fcmSwitchV24State** API, or wait for the **ADL_FCM_EVENT_V24_DATA_MODE_EXT** event. This one will come when the external application sends the "ATO" command: the serial link is switched to data mode, and then all V24 clients are notified.
- When a GSM data call is released from the remote part, the GSM flow will automatically be unsubscribed (the ADL_FCM_EVENT_FLOW_CLOSED event will be received by all the flow subscribers).
- When a GPRS session is released, or when a GSM data call is released from the Wireless CPU® side (with the adl_callHangUp function), the corresponding GSM or GPRS flow have to be unsubscribed. These flows will have to be subscribed again before starting up a new GSM data call, or a new GPRS session.
- For serial link flows, the serial line parameters (speed, character framing, etc...) must not be modified while the flow is in data state. In order to change these parameters' value, the concerned flow has firstly to be switched back in AT mode with the **adl_fcmSwitchV24State** API. Once the parameters changed, the flow may be switched again to data mode, using the same API.

3.9.4 The adl_fcmUnsubscribe Function

This function unsubscribes from a previously subscribed FCM service, closing the previously opened flows. The unsubscription will be effective only when the control event handler has received the **ADL_FCM_EVENT_FLOW_CLOSED** event.

If slave handles were subscribed, as soon as the master one unsubscribes from the flow, all the slave one will also be unsubscribed.

- Prototype

```
s8    adl_fcmUnsubscribe     (u8    Handle );
```

- Parameters

  Handle:

  Handle returned by the **adl_fcmSubscribe** function.

- Returned values
  - **OK** on success. The Control handler will also receive a **ADL_FCM_EVENT_FLOW_CLOSED** event when flow is ready to process
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is incorrect,
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if the flow is already unsubscribed,
  - **ADL_RET_ERR_BAD_STATE** if the serial link is not in AT mode.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.9.5 The adl_fcmReleaseCredits Function

This function releases some credits for requested flow handle.

The slave subscribers should not use this API.

- Prototype

```
s8    adl_fcmReleaseCredits   (     u8    Handle,
                                    u8    NbCredits );
```

- Parameters

  Handle:

  Handle returned by the **adl_fcmSubscribe** function.

  NbCredits:

  Number of credits to release for this flow. If this number is greater than the number of previously received data blocks, all credits are released. If an application wants to release all received credits at any time, it should call the **adl_fcmReleaseCredits** API with NbCredits parameter set to 0xFF.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
  - **ADL_RET_ERR_BAD_HDL** if the handle is a slave one.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.9.6 The adl_fcmSwitchV24State Function

This function switches a serial link state to AT mode or to Data mode. The operation will be effective only when the control event handler has received an **ADL_FCM_EVENT_V24_XXX_MODE** event. Only the main handle subscriber can use this API.

- Prototype

```
s8    adl_fcmSwitchV24State   (     u8    Handle,
                                    u8    V24State );
```

- Parameters

Handle:

Handle returned by the **adl_fcmSubscribe** function.

V24State:

Serial link state to switch to. Allowed values are defined below:

ADL_FCM_V24_STATE_AT,

ADL_FCM_V24_STATE_DATA.

- Returned values
  - **OK** on success. The Control handler will also receive a **ADL_FCM_EVENT_V24_XXX_MODE** event when the serial link state has changed
  - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown
  - **ADL_RET_ERR_BAD_HDL** if the handle is not the main flow one
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.9.7 The adl_fcmSendData Function

This function sends a data block on the requested flow.

- Prototype

```
s8 adl_fcmSendData     (     u8    Handle,
                             u8 *  Data,
                             u16   DataLen );
```

- Parameters

Handle:

Handle returned by the **adl_fcmSubscribe** function.

Data:

Data block buffer to write.

DataLen:

Data block buffer size.

Maximum data packet size depends on the subscribed flow:

  - On serial link based flows: 2000 bytes ;
  - On GSM data flow: no limitation (memory allocation size) ;
  - On GPRS flow: 1500 bytes ;
  - On Bluetooth virtual ports: 2000 bytes.

- Returned values
  - **OK** on success. The Control handler will also receive a **ADL_FCM_EVENT_MEM_RELEASE** event when the data block memory buffer will be released ;
  - **ADL_FCM_RET_OK_WAIT_RESUME** on success, but the last credit was used. The Control handler will also receive a **ADL_FCM_EVENT_MEM_RELEASE** event when the data block memory buffer will be released ;
  - **ADL_RET_ERR_PARAM** is a parameter has an incorrect value,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
  - **ADL_RET_ERR_BAD_STATE** if the flow is not ready to send data,
  - **ADL_FCM_RET_ERR_WAIT_RESUME** if the flow has no more credit to use.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).
  - On **ADL_FCM_RET_XXX_WAIT_RESUME** returned value, the subscriber has to wait for a **ADL_FCM_EVENT_RESUME** event on Control Handler to continue sending data.

3.9.8 The adl_fcmSendDataExt Function

This function sends a data block on the requested flow. This API do not perform any processing on provided data block, which is sent directly on the flow.

- Prototype

```
s8 adl_fcmSendDataExt   (     u8                      Handle,
                              adl_fcmDataBlock_t *    DataBlock);
```

- Parameters

  Handle:

  Handle returned by the **adl_fcmSubscribe** function.

  DataBlock:

  Data block buffer to write, using the following type:

```
typedef struct
{
   u16 Reserved1[4];
   u32 Reserved3;
   u16 DataLength;      /* Data length */
   u16 Reserved2[5];
   u8  Data[1];         /* Data to send */
} adl_fcmDataBlock_t;
```

The block must be dynamically allocated and filled by the application, before sending it to the function. The allocation size has to be **sizeof (adl_fcmDataBlock_t) + DataLength**, where DataLength is the value to be set in the **DataLength** field of the structure.

Maximum data packet size depends on the subscribed flow :

- On serial link based flows : 2000 bytes ;
- On GSM data flow : no limitation (memory allocation size) ;
- On GPRS flow : 1500 bytes ;
- On Bluetooth virtual ports : 2000 bytes.

- Returned values
  - **OK** on success. The Control handler will also receive a **ADL_FCM_EVENT_MEM_RELEASE** event when the data block memory buffer will be released,
  - **ADL_FCM_RET_OK_WAIT_RESUME** on success, but the last credit was used. The Control handler will also receive a **ADL_FCM_EVENT_MEM_RELEASE** event when the data block memory buffer will be released ;
  - **ADL_RET_ERR_PARAM** is a parameter has an incorrect value,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
  - **ADL_RET_ERR_BAD_STATE** if the flow is not ready to send data,
  - **ADL_FCM_RET_ERR_WAIT_RESUME** if the flow has no more credit to use.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).
  - On **ADL_FCM_RET_XXX_WAIT_RESUME** returned value, the subscriber has to wait for an **ADL_FCM_EVENT_RESUME** event on Control Handler to continue sending data.

Important Remark:

The Data block will be released by the **adl_fcmSendDataExt** API on OK and **ADL_FCM_RET_OK_WAIT_RESUME** return values (the memory buffer will be effectively released once the **ADL_FCM_EVENT_MEM_RELEASE** event will be received in the Control Handler). The application has to use only dynamic allocated buffers (with **adl_memGet** function).

3.9.9 The adl_fcmGetStatus Function

This function gets the buffer status for requested flow handle, in the requested way.

- Prototype

```
s8 adl_fcmGetStatus (   u8                  Handle,
                        adl_fcmWay_e        Way );
```

- Parameters

Handle:

Handle returned by the **adl_fcmSubscribe** function.

Way:

As flows have two ways (from Embedded application, and to Embedded application), this parameter specifies the direction (or way) from which the buffer status is requested. The possible values are:

```
typedef enum {
      ADL_FCM_WAY_FROM_EMBEDDED,
      ADL_FCM_WAY_TO_EMBEDDED
} adl_fcmWay_e;
```

- Returned values
  - **ADL_FCM_RET_BUFFER_EMPTY** if the requested flow and way buffer is empty,
  - **ADL_FCM_RET_BUFFER_NOT_EMPTY** if the requested flow and way buffer is not empty ; the Flow Control Manager is still processing data on this flow,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
  - **ADL_RET_ERR_PARAM** if the way parameter value in out of range.

3.10 GPIO Service

ADL provides a GPIO service to handle GPIO operations.

The defined operations are:

- A **`adl_ioGetCapabilitiesList`** function to retrieve a list of GPIO capablities informations.
- A **`adl_ioSubscribe`** function to set the reserved GPIO parameters
- A **`adl_ioUnsubscribe`** function to un-subscribes from a previously allocated GPIO handle
- A **`adl_ioEventSubscribe`** function to provide ADL with a call-back for GPIO related events
- A **`adl_ioEventUnsubscribe`** function to unsubscribe from the GPIO events notification
- A **`adl_ioSetDirection`** function to allow the direction of one or more previously allocated GPIO to be modified
- A **`adl_ioRead`** function to allow several GPIOs to be read from a previously allocated handle
- A **`adl_ioReadSingle`** function to allow one GPIO to be read from a previously allocated handle
- A **`adl_ioWrite`** function to write on several GPIOs from a previously allocated handle
- A **`adl_ioWriteSingle`** function to allow one GPIO to be written from a previously allocated handle

3.10.1 Required Header File

The header file for the GPIO functions is:

`adl_gpio.h`

3.10.2 GPIO Types

3.10.2.1 The adl_ioCap_t structure

This structure gives information about io capabilities.

```
typedef struct
{
 u32  NbGpio;    // The number of GPIO managed by ADL.
 u32  NbGpo;     // The number of GPO managed by ADL.
 u32  NbGpi;     // The number of GPI managed by ADL.
} adl_ioCap_t;
```

3.10.2.2 The adl_ioDefs_t type

This type defines the GPIO label.

This is a bit field:

- b0-b15 are use to identify the io
  - see **adl_ioLabel_e** type, section 3.10.2.4
- b16-b31 usage depends of the command
  - see **adl_ioLevel_e** type, section 3.10.2.4
  - see **adl_ioDir_e** type, section 3.10.2.5
  - see **adl_ioStatus_e** type, section 3.10.2.9
  - see **adl_ioCap_e** type, section 3.10.2.7
  - see **adl_ioError_e** type, section 3.10.2.6

3.10.2.3 The adl_ioLabel_e type

This type lists the label field definition (b0-b15 of **adl_ioDefs_t**). Each IO is identified by a number and a type. Please see also **adl_ioDefs_t** (section 3.10.2.1) for the other fields.

- Code

```
type def enum
{
   ADL_IO_NUM_MSK              = (0xFFF),
   ADL_IO_TYPE_POS             = 12,
   ADL_IO_TYPE_MSK             = (3UL<<ADL_IO_TYPE_POS),
   ADL_IO_GPI                  = (1UL<<ADL_IO_TYPE_POS),
   ADL_IO_GPO                  = (2UL<<ADL_IO_TYPE_POS),
   ADL_IO_GPIO                 = (3UL<<ADL_IO_TYPE_POS),
  _IO_LABEL_MSK                = ADL_IO_NUM_MSK | ADL_IO_TYPE_MSK
} adl_ioLabel_e
```

- Description

ADL_IO_NUM_MSK	Number field (b0-b11; 0->4095)
ADL_IO_TYPE_MSK	Type field (b12-b13):
ADL_IO_GPI	- To identify a GPI
ADL_IO_GPO	- To identify a GPO
ADL_IO_GPIO	- To identify a GPIO (GPO + GPI)
ADL_IO_LABEL_MSK	Mask including ADL_IO_NUM_MSK and ADL_IO_TYPE_MSK

Note:

b14-b15 are reserved.

3.10.2.4 The adl_ioLevel_e type

This type lists the level field definition (b16 of **adl_ioDefs_t**). Please see also **adl_ioDefs_t** (section 3.10.2.1) for the other fields.

- Code

```
type def enum
{
      ADL_IO_LEV_POS          = 16,
      ADL_IO_LEV_MSK          = (1UL<<ADL_IO_LEV_POS),
      ADL_IO_LEV_HIGH         = (1UL<<ADL_IO_LEV_POS),
      ADL_IO_LEV_LOW          = (0UL<<ADL_IO_LEV_POS)
} adl_ioLabel_e
```

- Description

ADL_IO_LEV_MSK	Level field: the Level of GPIO
ADL_IO_LEV_HIGH	- High Level
ADL_IO_LEV_LOW	- Low Level

3.10.2.5 The adl_ioDir_e type

This type lists the direction field definition (b17-b18 of **adl_ioDefs_t**). Please see also **adl_ioDefs_t** (section 3.10.2.1) for the other fields.

- Code

```
type def enum
{
      ADL_IO_DIR_POS          = 17,
      ADL_IO_DIR_MSK          = (3UL<<ADL_IO_DIR_POS),
      ADL_IO_DIR_OUT          = (0UL<<ADL_IO_DIR_POS),
      ADL_IO_DIR_IN           = (1UL<<ADL_IO_DIR_POS),
      ADL_IO_DIR_TRI          = (2UL<<ADL_IO_DIR_POS)
} adl_ioDir_e type
```

- Description

`ADL_IO_DIR_MSK`	- Dir field: The direction of GPIO
`ADL_IO_DIR_OUT`	- Set as Output
`ADL_IO_DIR_IN`	- Set as Input
`ADL_IO_DIR_TRI`	- Set as a Tristate

3.10.2.6 The adl_ioError_e type

This type lists the error field definition (b28-b31 of **adl_ioDefs_t**). Please see also **adl_ioDefs_t** (section 3.10.2.1) for the other fields.

- Code

```
type def enum
{
      ADL_IO_ERR_POS        = 28,
      ADL_IO_ERR_MSK        = (7UL<<ADL_IO_ERR_POS),
      ADL_IO_ERR            = (0UL<<ADL_IO_ERR_POS),
      ADL_IO_ERR_UNKWN      = (1UL<<ADL_IO_ERR_POS),
      ADL_IO_ERR_USED       = (2UL<<ADL_IO_ERR_POS),
      ADL_IO_ERR_BADDIR     = (3UL<<ADL_IO_ERR_POS),
      ADL_IO_ERR_NIH        = (4UL<<ADL_IO_ERR_POS),
      ADL_IO_GERR_POS       = 31,
      ADL_IO_GERR_MSK       = (1UL<<ADL_IO_GERR_POS),
      ADL_IO_GNOERR         = (0UL<<ADL_IO_GERR_POS),
      ADL_IO_GERR           = (1UL<<ADL_IO_GERR_POS)
 } ioError_e type
```

- Description

`ADL_IO_ERR_MSK`	Error cause (b28-b30):
`ADL_IO_ERR`	- Unidentified error
`ADL_IO_ERR_UNKWN`	- Unknown GPIO
`ADL_IO_ERR_USED`	- Already used
`ADL_IO_ERR_BADDIR`	- Bad direction
`ADL_IO_ERR_NIH`	- GPIO is not in the handle
`ADL_IO_GERR_MSK`	General error field (b31):
`ADL_IO_GNOERR`	- No Error (b28-30 are unsignificant)
`ADL_IO_GERR`	- Error during the treatment (see b28-b30 for the cause)

3.10.2.7 The adl_ioCap_e type

This type lists the capabilities field definition (b21-b22 of **adl_ioDefs_t**). It is only an output. Please see also **adl_ioDefs_t** (section 3.10.2.1) for the other fields.

- Code

```
type def enum
{
     ADL_IO_CAP_POS     = 21,
     ADL_IO_CAP_MSK     = (3UL<<ADL_IO_CAP_POS),
     ADL_IO_CAP_OR      = (1UL<<ADL_IO_CAP_POS),
     ADL_IO_CAP_IW      = (2UL<<ADL_IO_CAP_POS)
 } adl_ioCap_e type
```

- Description

ADL_IO_CAP_MSK	Capabilities field: Specials capabilities
ADL_IO_CAP_OR	- Output is readable
ADL_IO_CAP_IW	- Input is writable

3.10.2.8 The adl_ioStatus_e type

This type lists the status field definition (b19-b20 of **adl_ioDefs_t**). it is only an output. Please see also **adl_ioDefs_t** (section 3.10.2.1) for the other fields.

- Code

```
type def enum
{
      ADL_IO_STATUS_POS      = 19,
      ADL_IO_STATUS_MSK      = (3UL<<ADL_IO_STATUS_POS),
      ADL_IO_STATUS_USED     = (1UL<<ADL_IO_STATUS_POS),
      ADL_IO_STATUS_FREE     = (0UL<<ADL_IO_STATUS_POS)
} adl_ ioStatus_e type
```

- Description

ADL_IO_STATUS_MSK	Status field: to get the status of the fields
ADL_IO_STATUS_USED	- The IO is used by task
ADL_IO_STATUS_FREE	- The IO is available

3.10.2.9 The adl_ioEvent_e type

This type describes the GPIOs events received.

- Code

```
type def enum
{
      ADL_IO_EVENT_INPUT_CHANGED     = 2
} adl_ ioEvent_e type
```

- Description

ADL_IO_EVENT_INPUT_CHANGED	One or several of the subscribed inputs have changed. This event will be received only if a polling process is required at GPIO subscription time.

3.10.3 The adl_ioGetCapabilitiesList Function

This function returns the Wireless CPU® GPIO capabilities list. For each hardware available GPIO, the Wireless CPU® shall add an item in the GPIO capabilities list. A GPIO is hardware available when it is not used by any feature.

Caution:

The returned GpioTab array must be released by the customer application when the information is not useful any more.

- Prototype

```
s32 adl_ioGetCapabilitiesList  (u32 *             GpioNb,
                                adl_ioDefs_t **   GpioTab,
                                adl_ioCap_t *     GpioTypeNb );
```

- Parameters

GpioNb:

Number of GPIO treated, it is the size of **GpioTab** array.

GpioTab:

Returns a pointer to a list containing GPIO capablities informations (using **adl_ioDefs_t **** type).

Outputs available for each array element:

- o the GPIO label (see **adl_ioLabel_e** section 3.10.2.3).
- o the GPIO direction (see **adl_ioDir_e** section 3.10.2.5).
- o the GPIO capabilities (see **adl_ioCap_e** section 3.10.2.1).
- o the GPIO status (see **adl_ioStatus_e** section 3.10.2.9)

GpioTypeNb:

Returned the number of each GPIO, GPO and GPI. GpioTypeNb is an optional parameter, not used if set to NULL.

- Returned values
    - **OK** on success.
    - A negative error value otherwise:
        - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.

3.10.4 The adl_ioEventSubscribe Function

This function allows the Open AT® application to provide ADL with a call-back for GPIO related events.

- Prototype

```
s32 adl_ioEventSubscribe ( adl_ioHdlr_f GpioEventHandler );
```

- Parameters

  GpioEventHandler:

  Application provided event call-back function. Please refer to next chapter for event descriptions.

- Returned values
    - A positive or null value on success:
        - GPIO event handle, to be used on further GPIO API functions calls;
    - A negative error value otherwise:
        - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value,
        - **ADL_RET_ERR_NO_MORE_HANDLES** if the GPIO event service has been subscribed to more than 128 timers.
        - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level Interrupt handler.

Note:

In order to set-up an automatic GPIO polling process, the **adl_ioEventSubscribe** function has to be called before the **adl_ioSubscribe**.

3.10.5 The adl_ioHdlr_f Call-back Type

Such a call-back function has to be provided to ADL through the **adl_ioEventSubscribe** interface, in order to receive GPIO related events.

- Prototype

```
typedef void (*adl_ioHdlr_f) (s32            GpioHandle,
                              adl_ioEvent_e  Event,
                              u32            Size,
                              void *         Param );
```

- Parameters

GpioHandle:

Read GPIO handle for the **ADL_IO_EVENT_INPUT_CHANGED** event.

Event:

Event is the received identifier; other parameters use depends on the event type.

Size:

Number of items (read inputs or updated features) in the **Param** table.

Param:

Read value tables (using **adl_ioDefs_t *** type) for the **ADL_IO_EVENT_INPUT_CHANGED** event.

Outputs available for each array element:

- o the GPIO label (see **adl_ioLabel_e** 3.10.2.3).
- o the GPIO level (see **adl_ioLevel_e** 3.10.2.4).
- o the GPIO error information (see **adl_ioError_e** 3.10.2.6).

3.10.6 The adl_ioEventUnsubscribe Function

This function allows the Open AT® application to unsubscribe from the GPIO events notification.

- Prototype

```
s32 adl_ioEventUnsubscribe ( s32 GpioEventHandle );
```

- Parameters

GpioEventHandle:

Handle previously returned by the **adl_ioEventSubscribe** function.

- Returned values
  - A **OK** on success
  - A negative error value otherwise:
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown,
    - **ADL_RET_ERR_NOT_SUBSCRIBED** if no GPIO event handler has been subscribed,
    - **ADL_RET_ERR_BAD_STATE** if a polling process is currently running with this event handle.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

* Example:

```
void my_ioGetCapabilitiesList ()
    {
        u32 My_Loop;
        ascii * My_Message = adl_memGet ( 100 );
        u32 My_GpioNb;
        adl_ioDefs_t * My_GpioTab = NULL;
        adl_ioCap_t GpioTypeNb;

        adl_ioGetCapabilitiesList ( &My_GpioNb , &My_GpioTab ,
        &GpioTypeNb );

        wm_sprintf ( My_Message , "\r\nRessources : %d GPIO, %d GPI and
        %d GPO \r\n" , GpioTypeNb.NbGpio , GpioTypeNb.NbGpi ,
        GpioTypeNb.NbGpo );
        adl_atSendResponse ( ADL_AT_UNS, My_Message );

        adl_atSendResponse ( ADL_AT_UNS, "\r\nList of GPIO :\r\n" );

        for ( My_Loop = 0 ; My_Loop < My_GpioNb ; My_Loop++ )
        {
            switch ( My_GpioTab [ My_Loop ] & ADL_IO_TYPE_MSK )
            {
                case ADL_IO_GPI :
                    wm_sprintf ( My_Message, "GPI %d \r\n",
                    ( My_GpioTab [ My_Loop ] & ADL_IO_NUM_MSK ) );
                    break;
                case ADL_IO_GPIO :
                    wm_sprintf ( My_Message, "GPIO %d \r\n",
                    ( My_GpioTab [ My_Loop ] & ADL_IO_NUM_MSK ) );
                    break;
                case ADL_IO_GPO :
                    wm_sprintf ( My_Message, "GPO %d \r\n",
                    ( My_GpioTab [ My_Loop ] & ADL_IO_NUM_MSK ) );
                    break;
            }
            adl_atSendResponse ( ADL_AT_UNS, My_Message );

            ... // customer treatment

        }

        adl_memRelease ( My_Message );

        // My_GpioTab must be released by the customer application
        adl_memRelease ( My_GpioTab );
    }
```

3.10.7 The adl_ioSubscribe Function

This function subscribes to some GPIOs. For subscribed inputs, a polling system can be configured in order to notify a previously subscribed GPIO event handler with an **ADL_IO_EVENT_INPUT_CHANGED** event.

- Prototype

```
s32   adl_ioSubscribe   (     u32               GpioNb,
                              adl_ioDefs_t*     GpioConfig,
                              u8                PollingTimerType,
                              u32               PollingTime,
                              s32               GpioEventHandle );
```

- Parameters

GpioNb:

Size of the **GpioConfig** array.

GpioConfig:

GPIO subscription configuration array, which contains **GpioNb** elements. For each element, the **adl_ioDefs_t** structure members have to be configured.

- Inputs to set for each array element:
  - the label of the GPIO to subscribe (see **adl_ioLabel_e** section 3.10.2.3).
  - the GPIO direction (see **adl_ioDir_e** section 3.10.2.5).
  - the GPIO level, only if the GPIO is an output (see **adl_ioLevel_e** section 3.10.2.4.
- Outputs available for each array element:
  - the GPIO error information (see **adl_ioError_e** section 3.10.2.6).

PollingTimerType:

Type of the polling timer (if required); defined values are:

ADL_TMR_TYPE_100MS	100 ms granularity timer
ADL_TMR_TYPE_TICK	18.5 ms tick granularity timer

PollingTime:

If some GPIO are allocated as inputs, this parameter represents the time interval between two GPIO polling operations (unit is dependent on the **PollingTimerType** value).

Please note that each required polling process uses one of the available ADL timers (Reminder: up to 32 timers can be simultaneously subscribed).

If no polling is requested, this parameter has to be 0.

GpioEventHandle:

GPIO event handle (previously returned by **adl_ioEventSubscribe** function). Associated event handler will receive an **ADL_IO_EVENT_INPUT_CHANGED** event each time one of the subscribed inputs state has changed.

If no polling is requested, this parameter is ignored.

- Returned values
  - o A positive or null value on success:
    - GPIO handle to be used on further GPIO API functions calls;
  - o A negative error value otherwise (No GPIO is reserved):
    - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value,
    - **ADL_RET_ERR_DONE** refers to the field 3.10.2.6 **adl_ioError_e** for more information.
    - **ADL_RET_ERR_NO_MORE_TIMERS** if there is no timer available to start the polling process required by application,
    - **ADL_RET_ERR_NO_MORE_HANDLES** if no more GPIO handles are available.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.10.8 The adl_ioUnsubscribe Function

This function un-subscribes from a previously allocated GPIO handle.

- Prototype

```
s32    adl_ioUnsubscribe (      s32    GpioHandle );
```

- Parameters

GpioHandle:

Handle previously returned by **adl_ioSubscribe** function.

- Returned values
  - o A **OK** on success.
  - o A negative error value otherwise:
    - ADL_RET_ERR_UNKNOWN_HDL if the provided handle is unknown
    - ADL_RET_ERR_SERVICE_LOCKED if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.10.9 The adl_ioSetDirection Function

This function allows the direction of one or more previously allocated GPIO to be modified.

- Prototype

```
s32   adl_ioSetDirection (s32            GpioHandle,
                          u32            GpioNb,
                          adl_ioDefs_t*  GpioDir );
```

- Parameters

  GpioHandle:

  Handle previously returned by **adl_ioSubscribe** function.

  GpioNb:

  Size of the **GpioDir** array.

  GpioDir:

  GPIO direction configuration structure array (using the `adl_ioDefs_t *` type).

  - Inputs to set for each array element:
    - the label of the GPIO to modify (see **adl_ioLabel_e** section 3.10.2.3).
    - the new GPIO direction (see **adl_ioDir_e** section 3.10.2.5).
  - Outputs available for each array element:
    - the GPIO error information (see **adl_ioError_e** section 3.10.2.6)

- Returned values
  - **OK** on success.
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_DONE** refers to the field 3.10.2.6 **adl_ioError_e** for more information for each GPIO. If the error information is **ADL_IO_GNOERR**, the process has been completed with success for this GPIO.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.10.10 The adl_ioRead Function

This function allows several GPIOs to be read from a previously allocated handle.

- Prototype

```
s32   adl_ioRead  (s32              GpioHandle,
                   u32              GpioNb,
                   adl_ioDefs_t *   GpioRead );
```

- Parameters

  GpioHandle:

  Handle previously returned by **adl_ioSubscribe** function.

  GpioNb:

  Size of the **GpioRead** array.

  GpioRead:

  GPIO read structure array (using the **adl_ioDefs_t *** type).

  - Inputs to set for each array element:
    - the label of the GPIO to read (see **adl_ioLabel_e** section 3.10.2.3).
  - Outputs available for each array element:
    - the GPIO level value (see **adl_ioLevel_e** section 3.10.2.4).
    - the GPIO error information (see **adl_ioError_e** section 3.10.2.6)

- Returned values
  - **OK** on success (read values are updated in the **GpioArray** parameter).
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_DONE** refers to the field 3.10.2.6 **adl_ioError_e** for more information. If the error information is **ADL_IO_GNOERR**, the process has been completed with success for this GPIO.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.

3.10.11 The adl_ioReadSingle Function

This function allows one GPIO to be read from a previously allocated handle.

- Prototype

```
s32   adl_ioReadSingle  (s32                      GpioHandle,
                         adl_ioDefs_t             Gpio );
```

- Parameters

  GpioHandle:

  Handle previously returned by **adl_ioSubscribe** function.

  Gpio:

  Identifier of the GPIO (see **adl_ioLabel_e**).

- Returned values
  - GPIO read value on success (1 for a high level or 0 for a low level),
  - A negative error value otherwise
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown

- `ADL_RET_ERR_BAD_STATE` if one of the required GPIO was not subscribed as an input.

3.10.12 The adl_ioWrite Function

This function writes on several GPIOs from a previously allocated handle.

- Prototype

```
s32   adl_ioWrite (s32              GpioHandle,
                   u32              GpioNb,
                   adl_ioDefs_t *   GpioWrite );
```

- Parameters

GpioHandle:

Handle previously returned by **adl_ioSubscribe** function.

GpioNb:

Size of the **GpioWrite** array.

GpioWrite:

GPIO write structure array (using the adl_ioDefs_t * type).

- o Inputs to set for each array element:
  - the label of the GPIO to write (see **adl_ioLabel_e** section 3.10.2.3).
  - the new GPIO level (see **adl_ioLevel_e** section 3.10.2.4).
- o Outputs available for each array element:
  - the GPIO error information (see **adl_ioError_e** section 3.10.2.6).

- Returned values
  - o **OK** on success.
  - o A negative error value otherwise:
    - `ADL_RET_ERR_PARAM` if one parameter has an incorrect value.
    - `ADL_RET_ERR_DONE` refers to the field 3.10.2.6 **adl_ioError_e** for more information. If the error information is `ADL_IO_GNOERR`, the process has been completed with success for this GPIO.
    - `ADL_RET_ERR_UNKNOWN_HDL` if the handle is unknown.
    - `ADL_RET_ERR_BAD_STATE` if one of the required GPIOs was not subscribed as an output.

3.10.13 The adl_ioWriteSingle Function

This function allows one GPIO to be written from a previously allocated handle.

- Prototype

```
s32   adl_ioWriteSingle (s32              GpioHandle,
                         adl_ioDefs_t     Gpio,
                         bool             State );
```

- Parameters

  GpioHandle:

  Handle previously returned by **adl_ioSubscribe** function.

  Gpio:

  Identifier of the GPIO (see **adl_ioLabel_e**).

  State:

  Value to be set on the output:

  - TRUE for a high level.
  - FALSE for a low level.

- Returned values
  - **OK** on success.
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.
    - **ADL_RET_ERR_BAD_STATE** if one of the required GPIO was not subscribed as an input.

3.10.14 Example

This example demonstrates how to use the GPIO service in a nominal case (error cases not handled) on the Wireless CPU®.

Complete examples using the GPIO service are also available on the SDK (generic Telemetry sample, generic Drivers library sample).

```
// Global variables & constants

// Subscription data
#define GPIO_COUNT1 2
#define GPIO_COUNT2 1

const u32 My_Gpio_Label1 [ GPIO_COUNT1 ] = { 1 , 2 }
const u32 My_Gpio_Label2 [ GPIO_COUNT2 ] = { 3 }

const adl_ioDefs_t MyGpioConfig1 [ GPIO_COUNT1 ] =
{ { ADL_IO_GPIO | My_Gpio_Label1 [ 0 ] | ADL_IO_DIR_OUT | ADL_IO_LEV_LOW },
  { ADL_IO_GPIO | My_Gpio_Label1 [ 1 ] | ADL_IO_DIR_IN } };
 const adl_ioDefs_t MyGpioConfig2 [ GPIO_COUNT2 ] =
{ { ADL_IO_GPIO | My_Gpio_Label2 [ 0 ] | ADL_IO_DIR_IN } };


// Gpio Event Handle
s32 MyGpioEventHandle;

// Gpio Handles
s32 MyGpioHandle1, MyGpioHandle2;

// GPIO event handler
void MyGpioEventHandler ( s32 GpioHandle, adl_ioEvent_e Event, u32 Size,
void * Param )

{

     // Check event
      switch ( Event )
      {
         case ADL_IO_EVENT_INPUT_CHANGED :
         {
                u32 My_Loop;
                // The subscribed input has changed
                for ( My_Loop = 0 ; My_Loop < Size ; My_Loop++)
                {
                    if (( ADL_IO_TYPE_MSK & Param[ My_Loop ] )
                          && ADL_IO_GPO )
                    {
                        TRACE (( 1, "GPO %d new value: %d",
                        ( Param[ My_Loop ] ) & ADL_IO_NUM_MSK ,
                        ( Param[ My_Loop ]) & ADL_IO_LEV_MSK ) &&
                          ADL_IO_LEV_HIGH  ));
```

```
            }
            else
            {
                TRACE (( 1, "GPIO %d new value: %d",
                ( Param[ My_Loop ] ) & ADL_IO_NUM_MSK ,
                ( Param[ My_Loop ] ) & ADL_IO_LEV_MSK ) &&
                  ADL_IO_LEV_HIGH  ));
             }
          }
       }
       break;
   }
 }

    ...
// Somewhere in the application code, used as an event handler
    void MyFunction ( void )
    {
        // Local variables
        s32 ReadValue;

        // Subscribe to the GPIO event service
        MyGpioEventHandle = adl_ioEventSubscribe ( MyGpioEventHandler );

        // Subscribe to the GPIO service (One handle without polling,
        // one with a 100ms polling process)
        MyGpioHandle1 = adl_ioSubscribe ( GPIO_COUNT1, MyGpioConfig1, 0, 0,
0 );
        MyGpioHandle2 = adl_ioSubscribe ( GPIO_COUNT2, MyGpioConfig2,
        ADL_TMR_TYPE_100MS, 1, MyGpioEventHandle );

        // Set output
        adl_ioWriteSingle ( MyGpioHandle1, ADL_IO_GPIO | My_Gpio_Label1
        [ 0 ] , TRUE );

        // Read inputs
        ReadValue = adl_ioReadSingle (MyGpioHandle1, ADL_IO_GPIO |
        My_Gpio_Label1 [ 1 ] );
        ReadValue = adl_ioReadSingle (MyGpioHandle2, ADL_IO_GPIO |
        My_Gpio_Label2 [ 0 ] );

        // Unsubscribe from the GPIO services
        adl_ioUnsubscribe ( MyGpioHandle1 );
        adl_ioUnsubscribe ( MyGpioHandle2 );

        // Unsubscribe from the GPIO event service
        adl_ioEventUnsubscribe ( MyGpioEventHandle );
    }
```

3.11 Bus Service

The ADL supplies interface to handle bus operations.

The defined operations are:

- **adl_busSubscribe** to open a bus
- **adl_busUnsubscribe** to close a bus
- **adl_busIOCtl** to modify the behavior of the bus
- **adl_busRead** & **adl_busReadExt** to read on the a SPI or I2C bus
- **adl_busWrite** & **adl_busWriteExt** to write on the a SPI or I2C bus
- **adl_busDirectWrite** & **adl_busDirectRead** to write on the Parallel bus

3.11.1 Required Header File

The header file for the bus functions is:

```
adl_bus.h
```

3.11.2 Capabilities Registry Informations

3.11.2.1 The adl_busSpiCommonCap_e Type

SPI block common capabilities.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_COMMON_CAP_MASTER      = (1<<0),
 ADL_BUS_SPI_COMMON_CAP_SLAVE       = (1<<1),
 ADL_BUS_SPI_COMMON_CAP_2W          = (1<<2),
 ADL_BUS_SPI_COMMON_CAP_3W          = (1<<3),
 ADL_BUS_SPI_COMMON_PADDING         = 0x7fffffff
} adl_busSpiCommonCap_e;
```

- Description:

ADL_BUS_SPI_COMMON_CAP_MASTER	The block can be used in master mode.
ADL_BUS_SPI_COMMON_CAP_SLAVE	The block can be used in slave mode. **Reserved for future use.**
ADL_BUS_SPI_COMMON_CAP_2W	The block can be configured to use 2 wires (DAT and CLK).
ADL_BUS_SPI_COMMON_CAP_3W	The block can be configured to use 3 wires (MISO, MOSI and CLK).

3.11.2.2 The adl_busSpiCap_e Type

SPI block capabilities in Master or Slave mode.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_CAP_BUSY          = (1<<0),
 ADL_BUS_SPI_CAP_LOAD          = (1<<1),
 ADL_BUS_SPI_CAP_CS_NONE       = (1<<2),
 ADL_BUS_SPI_CAP_CS_GPIO       = (1<<3),
 ADL_BUS_SPI_CAP_CS_HARD       = (1<<4),
 ADL_BUS_SPI_CAP_MSB           = (1<<5),
 ADL_BUS_SPI_CAP_LSB           = (1<<6),
 ADL_BUS_SPI_CAP_MICROWIRE     = (1<<7),
 ADL_BUS_SPI_CAP_MASK          = (1<<8),
 ADL_BUS_SPI_CAP_SHIFT         = (1<<9),
 ADL_BUS_SPI_CAP_PADDING       = 0x7fffffff
} adl_busSpiCap_e;
```

- Description:

ADL_BUS_SPI_CAP_BUSY	The block can use a BUSY signal.
ADL_BUS_SPI_CAP_LOAD	The block can use a LOAD signal.
ADL_BUS_SPI_CAP_CS_NONE	The block can work without Chip Select.
ADL_BUS_SPI_CAP_CS_GPIO	The block can work with a GPIO as Chip Select.
ADL_BUS_SPI_CAP_CS_HARD	The block can work with a dedicated hardware pin as Chip Select.
ADL_BUS_SPI_CAP_MSB	The block can send data MSB first.
ADL_BUS_SPI_CAP_LSB	The block can send data LSB first.
ADL_BUS_SPI_CAP_MICROWIRE	The block can be used in Microwire mode.
ADL_BUS_SPI_CAP_MASK	The block has a mask possibility.
ADL_BUS_SPI_CAP_SHIFT	The block has a shift possibility.

3.11.2.3 The adl_busI2CCap_e Type

I2C block capabilities.

- Code:

```
typedef enum
{
 ADL_BUS_I2C_CAP_ADDR_10_BITS       = (1<<0),
 ADL_BUS_I2C_CAP_MASTER             = (1<<1),
 ADL_BUS_I2C_CAP_SLAVE              = (1<<2),
 ADL_BUS_I2C_CAP_CLK_FAST           = (1<<3),
 ADL_BUS_I2C_CAP_CLK_HIGH           = (1<<4),
 ADL_BUS_I2C_CAP_ADD_SIZE_8         = (1<<5),
 ADL_BUS_I2C_CAP_ADD_SIZE_16        = (1<<6),
 ADL_BUS_I2C_CAP_ADD_SIZE_24        = (1<<7),
 ADL_BUS_I2C_CAP_ADD_SIZE_32        = (1<<8),
 ADL_BUS_I2C_CAP_PADDING            = 0x7fffffff
} adl_busI2CCap_e;
```

- Description:

ADL_BUS_I2C_CAP_ADDR_10_BITS	The block can use 10 bits addressing mode.
ADL_BUS_I2C_CAP_MASTER	The block can be used in master mode.
ADL_BUS_I2C_CAP_SLAVE	The block can be used in slave mode. Reserved for future use.
ADL_BUS_I2C_CAP_CLK_FAST	The block can use Fast clock (400 kbits/s).
ADL_BUS_I2C_CAP_CLK_HIGH	The block can use High Speed clock (3.4 Mbits/s).
ADL_BUS_I2C_CAP_ADD_SIZE_8	The address size can be 8 bits (see **ADL_BUS_CMD_SET_ADD_SIZEe IOCtl** command).
ADL_BUS_I2C_CAP_ADD_SIZE_16	The address size can be 16 bits (see **ADL_BUS_CMD_SET_ADD_SIZE IOCtl** command).
ADL_BUS_I2C_CAP_ADD_SIZE_24	The address size can be 24 bits (see **ADL_BUS_CMD_SET_ADD_SIZE IOCtl** command).
ADL_BUS_I2C_CAP_ADD_SIZE_32	The address size can be 32 bits (see **ADL_BUS_CMD_SET_ADD_SIZE IOCtl** command).

3.11.3 Common Data Structures and Enumerations

ADL provides capabilities information about the BUS service, thanks to the registry service.

The following entries are defined in the registry:

Registry entry	Type	Description
i2c_NbBlocks[3]	INTEGER	The number of i2c blocks managed by the Wireless CPU®
i2c_xx_Cap	INTEGER	The capabilities of the block, defined as a combination of the **adl_busI2CCap_e** type values.
i2c_xx_MaxLength	Unsigned INTEGER[4]	The maximum amount of items that can be passed in a I2C read/write operation
spi_NbBlocks[3]	INTEGER	The number of spi blocks managed by the Wireless CPU®
spi_xx_Common	INTEGER	The generic capabilities of the block, defined as a combination of the **adl_busSpiCommonCap_e** type values.
spi_xx_ClockDivStep	INTEGER	The number of steps of the clock divider (see 3.11.2 **adl_busSPISettings_t::Clk_Speed** field description)
spi_xx_MaxLength	INTEGER	The maximum amount of items that can be passed in a SPI read/write operation
spi_xx_DataSizes[2]	INTEGER	Available data sizes for **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command
spi_xx_Master_OpcodeSizes[2]	Unsigned INTEGER[4]	Available Opcode sizes for **ADL_BUS_CMD_SET_OP_SIZE IOCtl** command
spi_xx_Master_AddressSizes[2]	Unsigned INTEGER[4]	Available Address sizes for **ADL_BUS_CMD_SET_ADD_SIZE IOCtl** command
spi_xx_Master_Cap	INTEGER	The capabilities of the block in Master mode, defined as a combination of the **adl_busSpiCap_e type**
spi_xx_Master_MaxFreqClock	INTEGER	The maximum frequency (in kHz) of the clock in Master mode (see 3.11.2 **adl_busSPISettings_t::Clk_Speed** field description).
Para_NbBlocks[3]	INTEGER	The number of parallel bus blocks managed by the Wireless CPU®
Para_NbCS	INTEGER	The number of chip select available to the customer

Registry entry	Type	Description
Para_CS	INTEGER	The list of currently accessible chip select * This is a bitfield, each bit represents a CS available. e.g. : Para_CS = 5, the Parallel bus 1 has 2 CS available : CS0 (b0) and CS2 (b2)
Para_xx_Addr	INTEGER	Current address of the Chip select XX
Para_xx_Freq	INTEGER	Current frequency of the Chip select XX

Note:

1. For the registry entry the **xx** part has to be replaced by the number of the instance.
   Example: if you want the capabilities of the I2C1 block the registry entry to use will be **i2c_01_Cap**. **Example:** if you want the common capabilities of the SPI2 block the registry entry to use will be **spi_02_Common**.
2. Sizes are coded in a bit field, where size n is available when the n-1 bit is set. Example: **0x80008003** means sizes 32 bits, 16 bits, 2 bits and 1 bit are available.
3. A SPI/I2C/Parallel bus block will be identified with a number from 1 to **spi_NbBlocks** or **i2c_NbBlocks** or **Parallel_NbBlocks**.
4. Entries using the Unsigned INTEGER type have to be casted to an u32 value after being retrieved from adl_regGetHWInteger function.

3.11.3.1 The _adl_busSettings_u Type

Generic bus settings union.

- Code

```
typedef struct
  {
      adl_busSPISettings_t    SPI;
      adl_busI2Settings_t     I2C;
  }adl_busSettings_u;
```

- Description

SPI

SPI member, previously handle SPI related settings.

I2C

I2C member, previously to handle 12C related settings.

3.11.3.2 The adl_busID_e Type

This type allows to identify the bus types supported by the service.

- Code:

```
typedef enum
{
 ADL_BUS_ID_SPI,       //SPI Bus
 ADL_BUS_ID_I2C,       //I2C Bus
 ADL_BUS_ID_PARALLEL,  //Parallel Bus
ADL_BUS_ID_LAST,       //Reserved for internal use
} adl_busID_e;
```

3.11.3.3 The adl_busType_e Type

Former enumeration used to identify BUS types.

- Code:

```
typedef enum
{
 ADL_BUS_SPI1,
 ADL_BUS_SPI2,
 ADL_BUS_I2C,
 ADL_BUS_PARALLEL,
} adl_busType_e;
```

- Description:

ADL_BUS_SPI1	This constant was previously used to access the Wireless CPU® SPI1 bus.
ADL_BUS_SPI2	This constant was previously used to access the Wireless CPU® SPI2 bus
ADL_BUS_I2C	This constant was previously used to access the Wireless CPU® I2C bus
ADL_BUS_PARALLEL	This constant was previously used to access the Wireless CPU® Parallel bus

3.11.4 SPI Bus Subscription Data Structures and Enumerations

3.11.4.1 The adl_busSPISettings_t Type

SPI bus settings.

- Code:

```
typedef struct
{
        u32             Clk_Speed;
        u32             Clk_Mode;
        u32             ChipSelect;
        u32             ChipSelectPolarity;
        u32             LsbFirst;
        adl_ioDefs_t    GpioChipSelect;
        u32             LoadSignal;
        u32             DataLinesConf;
        u32             MasterMode;
        u32             BusySignal;
} adl_busSPISettings_t;
```

- Description:

Clk_Speed

The Clk_Speed parameter allows to modify SPI bus clock speed.

Allowed values are in the [1 - N] range, where N is the spi_xx_ClockDivStep capability.

The SPI clock speed (in kHz) is defined using the formula below:

MaxFrequency / ClkSpeed

Where MaxFrequency is the Wireless CPU® maximum frequency for the current SPI block (spi_xx_Master_MaxFreqClock capability).

Example: if Clk_Speed is set to 1, and Max_Frequency is 13000 kHz, the SPI bus clock speed is set to 13000 kHz.

Clk_Mode

This parameter is the SPI clock mode (see 3.11.4.1 **adl_busSPI_Clk_Mode_e**).

ChipSelect

This parameter sets the pin used to handle the Chip Select signal (see 3.11.4.3 **adl_busSPI_ChipSelect_e**).

ChipSelectPolarity

This parameter sets the polarity of the Chip Select signal (see 3.11.4.4 **adl_busSPI_ChipSelectPolarity_e**).

LsbFirst

This parameter defines the priority for data transmission through the SPI bus, LSB or MSB first. This applies only to data. The Opcode and Address fields sent are always sent with MSB first (see 3.11.4.5 **adl_busSPI_LSBfirst_e**).

GpioChipSelect

This parameter defines the GPIO Chip Select. This parameter is used only if the ChipSelect parameter is set to the **ADL_BUS_SPI_ADDR_CS_GPIO** value.

LoadSignal

This parameter defines the LOAD signal behavior (see 3.11.4.7 **adl_busSPI_Load_e**).

DataLinesConf

This parameter defines if the SPI bus uses one single pin to handle both input and output data signals, or two pins to handle them separately (see 3.11.4.8 **adl_busSPI_DataLinesConf_e**).

MasterMode

This parameter is the SPI master or slave running mode (see 3.11.4.9 **adl_busSPI_MS_Mode_e**).

BusySignal

This parameter defines the LOAD signal behavior (see 3.11.4.10 **adl_busSPI_Busy_e**).

3.11.4.2 The adl_busSPI_Clk_Mode_e Type

SPI bus Clock Modes. See also 3.11.2 **adl_busSPISettings_t** for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_CLK_MODE_0,
 ADL_BUS_SPI_CLK_MODE_1,
 ADL_BUS_SPI_CLK_MODE_2,
 ADL_BUS_SPI_CLK_MODE_3,
 ADL_BUS_SPI_CLK_MODE_MICROWIRE,
} adl_busSPI_Clk_Mode_e;
```

- Description:

ADL_BUS_SPI_CLK_MODE_0	Mode 0: rest state 0, data valid on rising edge.
ADL_BUS_SPI_CLK_MODE_1	Mode 1: rest state 0, data valid on falling edge.
ADL_BUS_SPI_CLK_MODE_2	Mode 2: rest state 1, data valid on rising edge.
ADL_BUS_SPI_CLK_MODE_3	Mode 3: rest state 1, data valid on falling edge
ADL_BUS_SPI_CLK_MODE_MICROWIRE	Microwire mode. See also **ADL_BUS_SPI_CAP_MICROWIRE** Capability.

3.11.4.3 The adl_busSPI_ChipSelect_e Type

SPI bus Chip Select. See also 3.11.2 **adl_busSPISettings_t** for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_ADDR_CS_GPIO,
 ADL_BUS_SPI_ADDR_CS_HARD,
 ADL_BUS_SPI_ADDR_CS_NONE,
 } adl_busSPI_ChipSelect_e;
```

- Description:

ADL_BUS_SPI_ADDR_CS_GPIO	Use a GPIO as Chip Select signal (the GpioChipSelect parameter has to be used).
ADL_BUS_SPI_ADDR_CS_HARD	Use the reserved hardware chip select pin for the required bus.
ADL_BUS_SPI_ADDR_CS_NONE	The Chip Select signal is not handled by the ADL bus service. The application should allocate a GPIO to handle itself the Chip Select signal.

3.11.4.4 The adl_busSPI_ChipSelectPolarity_e Type

SPI bus Chip Select Polarity. See also 3.11.2 **adl_busSPISettings_t** for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_CS_POL_LOW,
 ADL_BUS_SPI_CS_POL_HIGH,
 } adl_busSPI_ChipSelectPolarity_e;
```

- Description:

ADL_BUS_SPI_CS_POL_LOW	Chip Select signal is active in Low state.
ADL_BUS_SPI_CS_POL_HIGH	Chip select signal is active in High state.

3.11.4.5 The adl_busSPI_LSBfirst_e Type

SPI bus MSB/LSB First. See also 3.11.2 **adl_busSPISettings_t** for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_MSB_FIRST,
 ADL_BUS_SPI_LSB_FIRST
 } adl_busSPI_LSBfirst_e;
```

- Description:

ADL_BUS_SPI_MSB_FIRST	Data buffer is sent with MSB first.
ADL_BUS_SPI_LSB_FIRST	Data buffer is sent with LSB first.

3.11.4.6 The adl_busSPI_WriteHandling_e Type

SPI bus Write Handling.

Kept for ascendant compatibility. The 3.11.4.7 **adl_busSPI_Load_e** type shall be used instead.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_FRAME_HANDLING,
 ADL_BUS_SPI_WORD_HANDLING
 } adl_busSPI_WriteHandling_e;
```

- Description:

ADL_BUS_SPI_FRAME_HANDLING	LOAD signal is enabled at the beginning of the read/write process, and is disabled at the end of this process.
ADL_BUS_SPI_WORD_HANDLING	LOAD signal state changes on each written or read word.

3.11.4.7 The adl_busSPI_Load_e Type

SPI bus LOAD signal configuration. See also 3.11.2 **adl_busSPISettings_t** & **ADL_BUS_SPI_CAP_LOAD** for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_LOAD_UNUSED,
 ADL_BUS_SPI_LOAD_USED
 } adl_busSPI_Load_e;
```

- Description:

ADL_BUS_SPI_LOAD_UNUSED	The LOAD signal is not used.
ADL_BUS_SPI_LOAD_USED	The LOAD signal is used (LOAD signal state changes on each written or read word; word size is defined thanks to **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command. Please refer to the PTS document for more information about the LOAD signal).

3.11.4.8 The adl_busSPI_DataLinesConf_e Type

SPI bus Data Lines configuration. See also 3.11.2 **adl_busSPISettings_t**, **ADL_BUS_SPI_COMMON_CAP_2W** & **ADL_BUS_SPI_COMMON_CAP_3W** capabilities for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_DATA_BIDIR,
 ADL_BUS_SPI_DATA_UNDIR
 } adl_busSPI_DataLinesConf_e;
```

- Description:

ADL_BUS_SPI_DATA_BIDIR	2 wires mode (DAT and CLK), one bi-directional pin is used to handle both input & output data signals.
ADL_BUS_SPI_DATA_UNDIR	3 wires mode (MISO, MOSI and CLK), two pins are used to handle separately input & output data signals.

3.11.4.9 The adl_busSPI_MS_Mode_e Type

Master/Slave bus mode configuration. See also 3.11.2 **adl_busSPISettings_t**, **ADL_BUS_SPI_COMMAN_CAP_MASTER** & **ADL_BUS_SPI_COMMAN_CAP_SLAVE** capabilities for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_MASTER_MODE,
 ADL_BUS_SPI_SLAVE_MODE
 } adl_busSPI_MS_Mode_e;
```

- Description:

ADL_BUS_SPI_MASTER_MODE	The SPI bus is running in master mode (default value when adl_busSubscribe function is used).
ADL_BUS_SPI_SLAVE_MODE	The SPI bus is running in slave mode. Reserved for future use.

3.11.4.10 The adl_busSPI_Busy_e Type

SPI bus BUSY signal configuration. See also 3.11.2 **adl_busSPISettings_t** & **ADL_BUS_SPI_CAP_BUSY** capability for more information.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_BUSY_UNUSED,
 ADL_BUS_SPI_BUSY_USED
 } adl_busSPI_Busy_e;
```

- Description:

ADL_BUS_SPI_BUSY_UNUSED	The BUSY signal is not used (default value when adl_busSubscribe function is used).
ADL_BUS_SPI_BUSY_USED	The BUSY signal is used

3.11.5 I2C Bus Subscription Data Structures and Enumerations

3.11.5.1 The adl_busI2CSettings_t Type

This structure defines the I2C bus settings for subscription.

Note:

Please refer to the Product Technical Specification for more information.

- Code:

```
typedef struct
{
       u32              ChipAddress;
       u32              Clk_Speed;
       u32              AddrLength;
       u32              MasterMode;
} adl_busI2CSettings_t;
```

- Description:

ChipAddress

This parameter sets the remote chip N bit address on the I2C bus.

Only b1 to bN bits are used (b0 bit and the most significant bytes are ignored).

N Value depends on the Wireless CPU® capabilities, and on the **adl_busI2CSettings_t::AddrLength** field configuration.

Example:

If the remote chip address is set to A0, the ChipAddress parameter has to be set to the 0xA0 value.

Clk_Speed

This parameter sets the required I2C bus speed (see 3.11.5.1 **adl_busI2C_Clk_Speed_e**).

AddrLength

This parameter sets the remote chip address length configuration (see 3.11.5.3 **adl_busI2C_AddrLength_e**).

MasterMode

This parameter is the I2C master or slave running mode (see 3.11.5.4 **adl_busI2C_MS_Mode_e**).

3.11.5.2 The adl_busI2C_Clk_Speed_e Type

I2C bus Clock Speed. See also 3.11.7.3 **adl_busI2CSettings_t**, **ADL_BUS_I2C_CAP_CLK_FAST** & **ADL_BUS_I2C_CAP_CLK_HIGH** capabilities for more information.

- Code:

```
typedef enum
{
 ADL_BUS_I2C_CLK_STD
 ADL_BUS_I2C_CLK_FAST
 ADL_BUS_I2C_CLK_HIGH
 } adl_busI2C_Clk_Speed_e;
```

- Description:

ADL_BUS_I2C_CLK_STD	Standard I2C bus speed (100 kbits/s).
ADL_BUS_I2C_CLK_FAST	Fast I2C bus speed (400 kbits/s).
ADL_BUS_I2C_CLK_HIGH	High I2C bus speed (3.4 Mbits/s).

3.11.5.3 The adl_busI2C_AddrLength_e Type

I2C bus chip address length. See also 3.11.7.3 **adl_busI2CSettings_t** & **ADL_BUS_I2C_CAP_ADDR_10BITS** capability for more information.

- Code:

```
typedef enum
{
 ADL_BUS_I2C_ADDR_7_BITS
 ADL_BUS_I2C_ADDR_10_BITS
 } adl_busI2C_AddrLength_e;
```

- Description:

ADL_BUS_I2C_ADDR_7_BITS	Chip address is 7 bits long (default value if adl_busSubscribe function is used).
ADL_BUS_I2C_ADDR_10_BITS	Chip address is 10 bits long.

3.11.5.4 The adl_busI2C_MS_Mode_e Type

Master/Slave bus mode configuration. See also 3.11.7.3 **adl_busI2CSettings_t** & **ADL_BUS_I2C_CAP_MASTER** capability for more information.

- Code:

```
typedef enum
{
 ADL_BUS_I2C_MASTER_MODE,
 ADL_BUS_I2C_SLAVE_MODE
 } adl_busI2C_MS_Mode_e;
```

- Description:

ADL_BUS_I2C_MASTER_MODE	The I2C bus is running in master mode (default value when **adl_busSubscribe** function is used).
ADL_BUS_I2C_SLAVE_MODE	The I2C bus is running in slave mode. Reserved for future use.

3.11.6 Parallel Bus Subscription Data Structures and Enumerations

3.11.6.1 The adl_busParallelCs_t Type

This type defines the Parallel bus Chip Select.

Please refer to the Product Technical Specification for more information.

- Code:

```
typedef struct
{
      u8          Type;        //Chip select type
      u8          Id;          //Chip select identifier
      u8          Pad[2];      //Needed to be compliant with GCC alignment
} adl_busParallelCs_t;
```

- Description:

  Type

  This parameter defines the Chip Select signal type.

  The only available value is ADL_BUS_PARA_CS_TYPE_CS. All other values are reserved for future use (see adl_busParallel_CS_Type_e).

  Id

  This parameter defines the Chip Select identifier used.

3.11.6.2 The adl_busParallelPageCfg_t Type

Configuration parameters for the page mode.

During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.

- Code:

```
typedef struct
{
        u8          PageSize;         //Page size
        u8          PageAccessCycles; //Between address change and valid
                                        data output
} adl_busParallelPageCfg_t;
```

3.11.6.3 The adl_busParallelSettings_t Type

Parallel bus settings.

- Code

```
typedef struct
{
        u8                                  Width;
        u8                                  Mode;
        u8                                  pad [2];
        adl_busParallelTimingsCfg_t         ReadCfg;
        adl_busParallelTimingsCfg_t         WriteCfg;
        adl_busParallelCs_t                 Cs;
        adl_busParallelPageCfg_t            PageCfg;
        adl_busParallelSynchronousCfg_t     SynchronousCfg;
        u32                                 AddressPin;
} adl_busSPISettings_t;
```

- Description:

  Width

  This parameter defines the read/write process data buffer items bit size, using the **adl_busParallelSize_e** type.

  Mode

  This parameter defines the required parallel bus standard mode to be used, using the **adl_busParallel_Bus_Mode_e** type.

  ReadCfg

  Define the timing configuration for each read and write process, using the **adl_busParallelTimingCfg_t** type.

WriteCfg

Define the timing configuration for each read and write process, using the **adl_busParallelTimingCfg_t** type.

Cs

Configuration parameters for the page mode.

During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.

PageCfg

Configuration parameters for the page mode.

During page modes access, other asynchronous mode read timings still apply. This structure hosts additional page-specific parameters.

SynchronousCfg

Configuration of the synchronous mode.

This structure hosts the parameters used to configure the synchronous mode accesses.

AddressPin

Select the pin used for the parallel bus. This is a bitfield, each bit represents a pin of the parrallel bus. e.g.: 0x03, two address pin are used (A0 and A1).

3.11.6.4 The adl_busParallelSynchronousCfg_t Type

Configuration parameters for the page mode.

This structure hosts the parameters used to configure the synchronous mode accesses.

- Code:

```
typedef struct
{
      u8          BurstSize;              //Size of Burst size
      u8          ClockDivisor;           //Main Memory clock divider
      s32         UseWaitEnable:1;        //WS generation using WAIT#
      s32         WaitActiveDuringWS:1;   //WAIT# during or 1-cycle before
                                            WS
      s32         Reserved:30;            //unused
   } adl_busParallelSynchronousCfg_t;
```

3.11.6.5 The adl_busParallelTimingCfg_t Type

Parallel bus Timing structure.

This type defines the Parallel bus timings.

Note:

The parameters configuration defines the parallel bus timing, in cycles number (please refer to the Product Technical Specification for more information), according to the bus mode required at subscription time (see **adl_busParallel_Bus_Mode_e**).

Example: In 26 MHz cycles number, one cycle duration is 1/26 MHz = ~38.5 ns

- Code

```
typedef struct
{
      u8          AccessTime;
      u8          SetupTime;
      u8          HoldTime;
      u8          TurnaroundTime;
      u8          OptoOpTurnaroundTime;
      u8          pad[3];                // Internal use only
} adl_busParallelTimingCfg_t;
```

- Description:

AccessTime

Access Time (see **adl_busParallel_Bus_Mode_e** and the Product Technical Specification).

SetupTime

Setup Time (see **adl_busParallel_Bus_Mode_e** and the Product Technical Specification).

HoldTime

Hold Time (see **adl_busParallel_Bus_Mode_e** and the Product Technical Specification).

TurnaroundTime

Turnaround Time (see **adl_busParallel_Bus_Mode_e** and the Product Technical Specification).

OptoOpTurnaroundTime

Read-to-read/write-to-write turnaround Time.

(see **adl_busParallel_Bus_Mode_e** and the Product Technical Specification)

3.11.6.6 The adl_busParallelSize_e Type

Bus access width.

Multiplexed modes spare pins by multiplexing data and addresses on the same pins. All the access widths and access modes are not available, valid combinations depend on the platform.

- Code

```
typedef enum
{
ADL_BUS_PARALLEL_WIDTH_INVALID,                  // reserved
ADL_BUS_PARALLEL_WIDTH_8_BITS,                   // 8-bit device
ADL_BUS_PARALLEL_WIDTH_16_BITS,                  // 16-bit device
ADL_BUS_PARALLEL_WIDTH_32_BITS,                  // 32-bit device
ADL_BUS_PARALLEL_WIDTH_16_BITS_MULTIPLEXED,      // 16-bit multiplexed
                                                 device
ADL_BUS_PARALLEL_WIDTH_32_BITS_MULTIPLEXED       //32-bit multiplexed
                                                 device
} adl_busParallelSize_e;
```

3.11.6.7 The adl_busParallel_Bus_Mode_e Type

Types of access.

Intel 8080 compatible and Motorola 6800 compatible asynchronous accesses modes can be configured:

- Intel mode uses an output enable or read enable signal and a write enable signal. In this read process example, Setup & Hold times are set to 1, and Access & Turnaround times are set to 3.

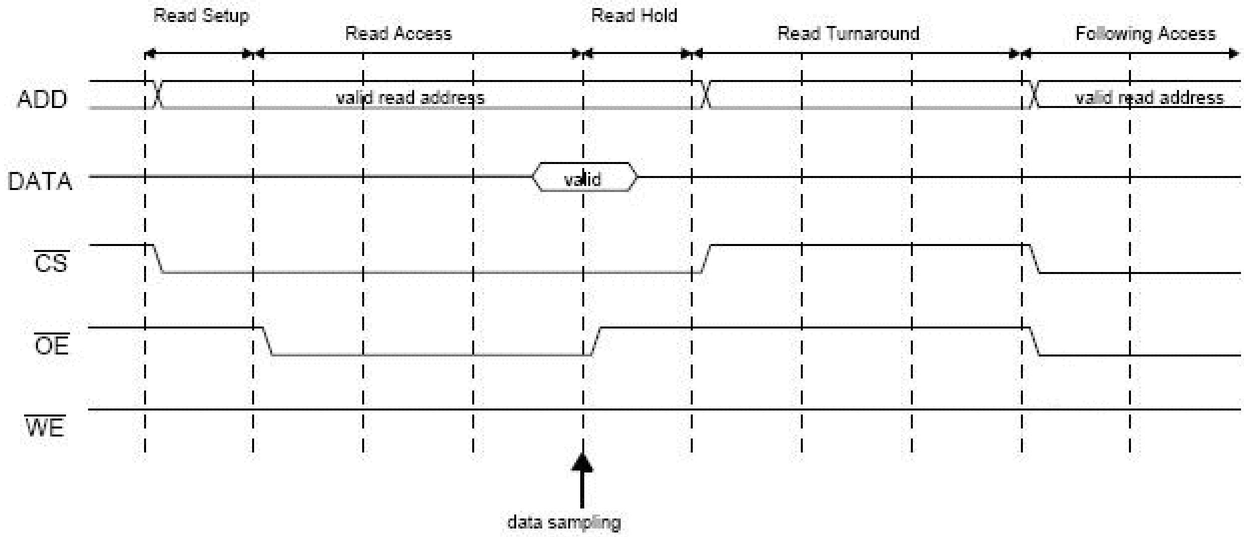


Figure 5: Intel Mode Timing - Read Process Example

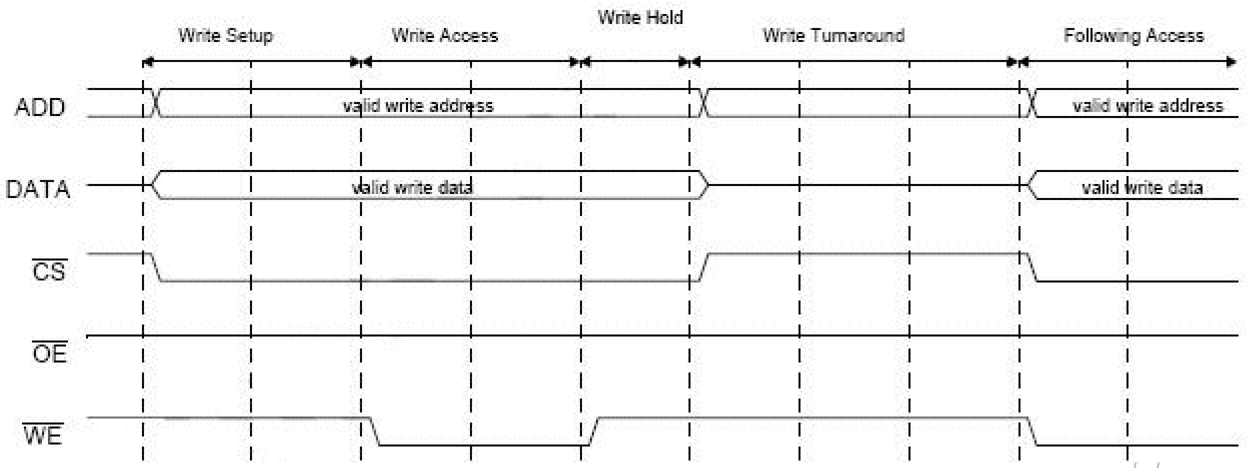


Figure 6: Intel Mode Timing - Write Process Example

- Motorola mode uses a read not write signal and an enable signal. The polarity of the enable signal can be configured:
  - o E is active at high level with mode Motorola 0 (LOW)
  - o E is active at low level with mode Motorola 1 (HIGH)

  The following timing behavior applies when the ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_LOW (E signal low polarity) or ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_HIGH (E signal high polarity) modes are required at subscription time. In the example given, the Access, Setup & Hold times are set to 1, and the Turnaround time is set to 2.

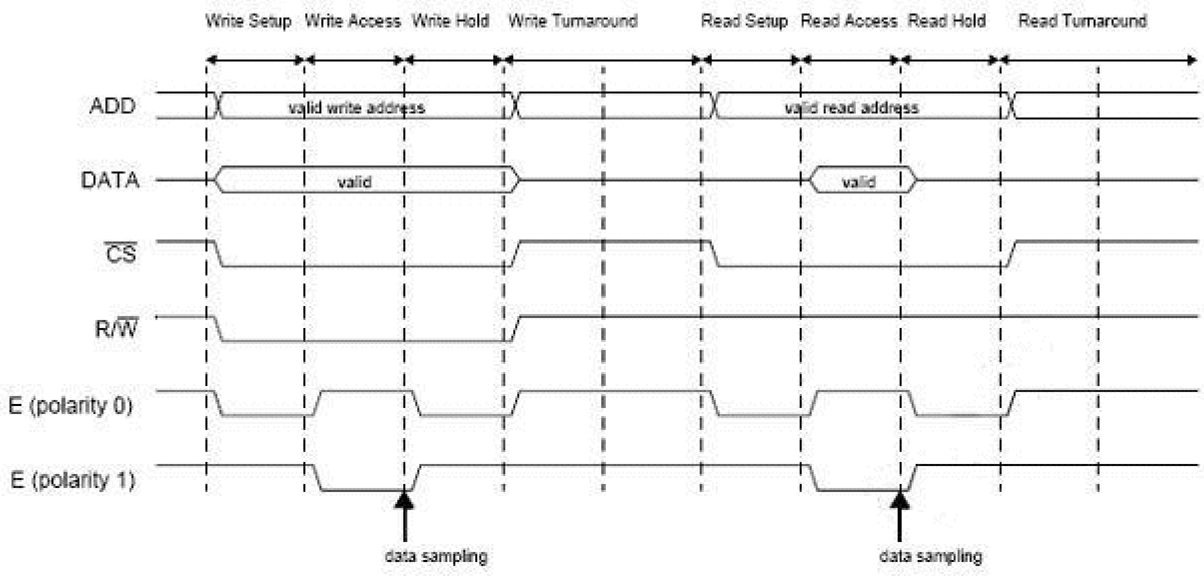


Figure 7: Motorola Modes Timing Example

- Code

```
enum
{
ADL_BUS_PARALLEL_MODE_INVALID,               // reserved
ADL_BUS_PARALLEL_MODE_ASYNC_INTEL,           // Intel 8080 compatible
ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_LOW,    // Motorola 6800
                                             compatible, with E signal
                                             low polarity
ADL_BUS_PARALLEL_MODE_ASYNC_MOTOROLA_HIGH,   // Motorola 6800
                                             compatible, with E signal
                                             high polarity
ADL_BUS_PARALLEL_MODE_ASYNC_PAGE,            // Page mode
ADL_BUS_PARALLEL_MODE_SYNC_READ_ASYNC_WRITE, // Synchronous only in
                                             reads
ADL_BUS_PARALLEL_MODE_SYNC_READ_WRITE        // Full synchronous mode
} adl_busParallel_Bus_Mode_e
```

3.11.6.8 The adl_busParallel_CS_Type_e Type

Parallel bus chip select type.

See also: **adl_busParallelCs_t** (section 3.11.6.1) for more informations.

- Code

```
enum
{
ADL_BUS_PARA_CS_TYPE_CS,     // Chip select type
} adl_busParallel_CS_Type_e
```

- Decription

The Type parameter defines the Chip Select signal type. The only available value is **ADL_BUS_PARA_CS_TYPE_CS**. All other values are reserved for future use.

3.11.7 IOCtl Operations Data Structures and Enumerations

3.11.7.1 The adl_busAsyncoInfo_t Type

This structure lists the information returned when an asynchronous read/write operation end event occurs.

- Code:

```
typedef struct
{
      s32              Result;
} adl_busAsyncoInfo_t;
```

- Description:

Result

Asynchronous read/write operation result code. See also **adl_busWrite** & **adl_busRead** functions return values description for more information.

3.11.7.2 The adl_busEvt_t Type

This structure allows to define the interrupt handlers which will be notified when the end of an asynchronous read/write operation event occurs.

Interrupt handlers defined in the IRQ service - using the adl_irqHandler_f type - are notified with the following parameters:

- the Source parameter will be set to **ADL_IRQ_ID_SPI_EOT** (for SPI bus operation) or **ADL_IRQ_ID_I2C_EOT** (for I2C bus operation).
- the **adl_irqEventData_t**::SourceData field of the Data parameter should be casted to the **adl_busAsyncInfo_t * type**, usable to retrieve information about the current interrupt event (if the **ADL_IRQ_OPTION_AUTO_READ** option has been required)
- the **adl_irqEventData_t**::Instance field of the Data parameter will have to be considered as an **u32** value, usable to identify which block has raised the current interrupt event (i.e. the BlockId provided at subscription time in adl_busSubscribe function).
- the **adl_irqEventData_t**::Context field of the Data parameter will be the application context, provided when the **adl_busReadExt** or **adl_busWriteExt** function was called. (It will be set to NULL if **adl_busRead** or **adl_busWrite** function was used)

- Code:

```
typedef struct
{
        s32                 LowLevelIrqHandle;
        s32                 HighLevelIrqHandle;
} adl_busEvt_t;
```

- Description:

LowLevelIrqHandle

Low level interrupt handler, previously returned by the **adl_irqSubscribe** function.

This parameter is optional if the **HighLevelIrqHandle** parameter is supplied.

HighLevelIrqHandle

High level interrupt handler, previously returned by the **adl_irqSubscribe** function.
This parameter is optional if the **LowLevelIrqHandle** parameter is supplied.

3.11.7.3 The adl_busSpiMaskShift_t Type

The parameter type for the **ADL_BUS_CMD_SET_SPI_MASK_AND_SHIFT** and **ADL_BUS_CMD_GET_SPI_MASK_AND_SHIFT IoCtl** commands.

- Code:

```
typedef struct
{
        u32                 w_Mask;
        u32                 w_Value;
        adl_busMaskSPI_e    Option;
        u8                  Pad [3];
} adl_busSpiMaskShift_t;
```

- Description:

w_Mask

Each bit to "1" will stay unchanged and each bit to "0" will be replaced by the w_Value ones.

w_Value

The value to set in the masked bits.

Option

Enabled/disabled Mask and Shift modes.

Pad

Internal use only.

3.11.7.4 The adl_busMaskSPI_e Type

Definition of the parameters to enable/disable Mask and Shift modes.

- Code:

```
typedef enum
{
 ADL_BUS_SPI_MASK_ENA                 = (1L<<0),
 ADL_BUS_SPI_SHIFT_ENA                = (1L<<1),
 } adl_busMaskSPI_e;
```

- Description:

ADL_BUS_SPI_MASK_ENA	Mask mode is enabled.
ADL_BUS_SPI_SHIFT_ENA	Shift mode is enabled.

3.11.7.5 The adl_busIoCtlCmd_e Type

Definition of the commands for adl_busIOCtl function.

- Code:

```
typedef enum
{
 ADL_BUS_CMD_SET_DATA_SIZE
 ADL_BUS_CMD_GET_DATA_SIZE
 ADL_BUS_CMD_SET_ADD_SIZE
 ADL_BUS_CMD_GET_ADD_SIZE
 ADL_BUS_CMD_SET_OP_SIZE
 ADL_BUS_CMD_GET_OP_SIZE
 ADL_BUS_CMD_LOCK
 ADL_BUS_CMD_UNLOCK
 ADL_BUS_CMD_GET_LAST_ASYNC_RESULT
 ADL_BUS_CMD_SET_ASYNC_MODE
 ADL_BUS_CMD_GET_ASYNC_MODE
 ADL_BUS_CMD_SET_SPI_MASK_AND_SHIFT
 ADL_BUS_CMD_GET_SPI_MASK_AND_SHIFT
 ADL_BUS_CMD_SET_PARALLEL_CFG
 ADL_BUS_CMD_GET_PARALLEL_CFG
 ADL_BUS_CMD_PARA_GET_ADDRESS
ADL_BUS_CMD_PARA_GET_MAX_SETTINGS
 ADL_BUS_CMD_PARA_GET_MIN_SETTINGS
 ADL_BUS_CMD_PADDING                        = 0x7fffffff
} adl_busIoCtlCmd_e;
```

- Description:

ADL_BUS_CMD_SET_DATA_SIZE	Set the size in bits of one data element. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an u32 value.

See also spi_xx_DataSizes Capability for the available values, default value is 8.

	<u>Note</u>: Available for the SPI Bus only.
ADL_BUS_CMD_GET_DATA_SIZE	Get the size in bits of one data element. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an u32 value. <u>Note</u>: Available for the SPI Bus only.
ADL_BUS_CMD_SET_ADD_SIZE	Set the size in bits of the address. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an u32 value. See also spi_xx_MasterAddressSizes and **adl_busI2CCap_e** capabilities for the available values, default value is zero (address is not used).
ADL_BUS_CMD_GET_ADD_SIZE	Set the size in bits of the address. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an u32 value. <u>Note</u>: Available for the SPI and I2C Bus only.
ADL_BUS_CMD_SET_OP_SIZE	Set the size in bits of the Opcode. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an u32 value. <u>Note</u>: Available for the SPI Bus only.
ADL_BUS_CMD_GET_OP_SIZE	Get the size in bits of the Opcode. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an u32 value. <u>Note</u>: Available for the SPI Bus only.
ADL_BUS_CMD_CLOCK	Lock a bus to avoid concurrent access and to allow access to the bus in interrupt context. After this call, the block is locked and only the handle which has locked it can use this block. Parameters:The **Param** of **adl_busIoCtl** is not relevant and can be set to NULL. <u>Note</u>: Available for the SPI and I2C Bus only. Trying to lock a second time a given block with the same handle will lead to an **ADL_RET_ERR_BAD_HDL** error. Trying to lock a bus which is already locked by another handle will lead the current task context to be suspended,

	until the block is unlocked, thanks to the **ADL_BUS_CMD_UNLOCK** command Warning: This command is available only in asynchronous mode.
ADL_BUS_CMD_UNLOCK	Unlock a bus previously locked by **ADL_BUS_CMD_LOCK** command. Parameters: The **Param** of **adl_busIoCtl** is not relevant and can be set to NULL. Note: Available for the SPI and I2C Bus only. If a task context was suspended due to a **ADL_BUS_CMD_LOCK** command on this block, it will be resumed as soon as the block is unlocked.
ADL_BUS_CMD_GET_LAST_ASYNC_RESULT	Get the last asynchronous read/write operation of return value. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an adl_busAsyncInfo_t structure. Note: Available for the SPI and I2C Bus only.
ADL_BUS_CMD_SET_ASYNC_MODE	Configure the Synchronous/asynchronous mode settings Parameters: The **Param** of **adl_busIOCtl** is defined as pointer on **adl_busEvt_t.** When this parameter is set to a value different of NULL, **adl_busWrite** and **adl_busRead** behaviour become asynchronous. When it is set to NULL, read/write operations are synchronous (default value). Note: Available for the SPI and I2C Bus only
ADL_BUS_CMD_GET_ASYNC_MODE	Get the current value of the synchronous/asynchronous mode settings. Parameters: The **Param** of **adl_busIOCtl** is defined as a pointer on **adl_busEvt_t.** If the current mode is synchronous, all elements of Param\ are NULL. Available for the SPI and I2C Bus only.
ADL_BUS_CMD_SET_SPI_MASK_AND_SHIFT	Enable/disable and set the parameters for the mask and shift modes. Parameters: The **Param** of **adl_busIOCtl** is defined as a pointer on **adl_busSpiMaskShift_t**.

	Note: Available for the SPI Bus only. Warning: Reserved for future use
ADL_BUS_CMD_GET_SPI_MASK_AND_SHIFT	Get the status and the parameters for the mask and shift modes. Parameters:The **Param** of **adl_busIOCtl** is defined as a pointer on **adl_busSpiMaskShift_t.** Note: Available for the SPI Bus only. Warning: Reserved for future use.
ADL_BUS_CMD_SET_PARALLEL_CFG	Set the Parallel configuration for one subscribed bus. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer on **adl_busParallelSettings_t**. Note: Available for the Parallel Bus only.
ADL_BUS_CMD_GET_PARALLEL_CFG	Get the Parallel configuration for one subscribed bus. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer on **adl_busParallelSettings_t**. Note: Available for the Parallel Bus only.
ADL_BUS_CMD_PARA_GET_ADDRESS	Gets Parallel bus base where the chip select can be addressed for one subscribed bus. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer to an u32. Note: Available for the Parallel Bus only.
ADL_BUS_CMD_PARA_GET_MAX_SETTINGS	Gets Parallel bus maximum values. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer on **adl_busParallelSettings_t**. Only the Width, the Mode, the ReadCfg, the WriteCfg and the SynchronousCfg informations are availables Note: Available for the Parallel Bus only.
ADL_BUS_CMD_PARA_GET_MIN_SETTINGS	Gets Parallel bus minimum values. Parameters: The **Param** of **adl_busIoCtl** is defined as a pointer on **adl_busParallelSettings_t**. Only the Width, the Mode, the ReadCfg, the WriteCfg and the SynchronousCfg informations are availables Note: Available for the Parallel Bus only.

3.11.8 Read/Write Data Structures

3.11.8.1 The adl_busAccess_t Type

This structure sets the bus access configuration parameters, to be used on a standard read or write process request (for SPI or I2C bus only).

- Code:

```
typedef struct
{
        u32                 Address;
        u32                 Opcode;
} adl_busAccess_t;
```

- Description

  Address

  The **Address** parameter allows up to 32 bits to be sent on the bus, before starting the read or write process. The number of bits to send is set by the **ADL_BUS_CMD_SET_ADD_SIZE** command. If less than 32 bits are required to be sent; only the most significant bits are sent on the bus.

  Opcode

  The **Opcode** parameter allows up to 32 bits to be sent on the bus, before starting the read or write process. The number of bits to send is set by the **ADL_BUS_CMD_SET_OP_SIZE** command. If less than 32 bits are required to be sent, only the most significant bits are sent on the bus. Usable only for SPI bus (ignored for I2C bus).

  Example: In order to send the "BBB" word on the bus prior to a read or write process, the Opcode parameter has to be set to the 0xBBB00000 value, and the OpcodeLength parameter has to be set to 12.

3.11.9 The adl_busSubscribe Function

This function subscribes to a specific bus, in order to write and read values to/from a remote chip.

- Prototype

```
s32    adl_busSubscribe  (adl_busID_e            BusId,
                          u32                    BlockId,
                          void *                 BusParam );
```

- Parameters

  BusId:

  Type of the bus to subscribe to, using the **adl_busID_e** type values.

  BlockId:

  ID of the block to use (in the range 1-N, where N is specific to each bus type & Wireless CPU® platform; cf. the i2c_NbBlocks & spi_NbBlocks & Para_NbBlocks Capabilities).

BusParam:

Subscribed bus configuration parameters, using specifics parameters of the bus (considered as an **adl_busSPISettings_t ***, **an adl_busI2CSettings_t *** or an **adl_busParallelSettings_t *** pointer).

- Returned values
  - o Handle: A positive or null value on success:
    - ▪ BUS handle, to be used in further BUS API functions calls;
  - o A negative error value:
    - ▪ **ADL_RET_ERR_PARAM** if a parameter has an incorrect value
    - ▪ **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the required bus is already subscribed with the provided configuration
    - ▪ **ADL_RET_ERR_BAD_HDL** if a GPIO required by the provided bus configuration is currently subscribed by an Open AT® application.
    - ▪ **ADL_RET_ERR_NOT_SUPPORTED** if the required bus type is not supported by the Wireless CPU® on which the application is running.
    - ▪ **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Notes:

A bus is available only if the GPIO multiplexed with the corresponding feature is not yet subscribed by an Open AT® application.

Once the bus is subscribed, the multiplexed GPIO with the required configuration are not available for subscription by the Open AT® application, or through the standard AT commands.

3.11.10 The adl_busUnsubscribe Function

This function unsubscribes from a previously subscribed.

- Prototype

```
s32    adl_busUnsubscribe (s32       Handle );
```

- Parameters

Handle:

Handle previously returned by the **adl_busSubscribe** function.

- Returned values
  - o **OK** on success.
  - o A negative error value otherwise.
    - ▪ **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
    - ▪ **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.11.11 The adl_busIOCtl Function

This function permits to modify the configuration and the behavior of a subscribed bus.

- Prototype

```
s32    adl_busIOCtl      (u32                      Handle,
                          adl_busIoCtlCmd_e        Cmd,
                          void *                   Param );
```

- Parameters

Handle:

Handle previously returned by the **adl_busSubscribe** function.

Cmd:

Command to be executed. (see 3.11.7.5 **adl_busIoCtlcmd_e** for more information).

Param:

Parameter associated to the command. (see 3.11.7.5 **adl_busIoCtlcmd_e** for more information).

- Returned values
    - **OK** on success
    - A negative error value:
        - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value
        - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.
        - **ADL_RET_ERR_DONE** if an error occurs during the operation.
        - **ADL_RET_ERR_BAD_HDL** if the required command is not usable for the current handle.
        - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.11.12 The adl_busRead Function

This function reads data from a previously subscribed bus SPI or I2C type.

<u>Note:</u>

By default the access is synchronous. This behavior can be changed with the **ADL_BUS_CMD_SET_ASYNC_MODE IOCtl** command.

- Prototype

```
s32    adl_busRead (s32                  Handle,
                    adl_busAccess_t *    pAccessMode,
                    u32                  Length,
                    void *               pDataToRead );
```

- Parameters

  Handle:

  Handle previously returned by the **adl_busSubscribe** function.

  pAccessMode:

  Bus access mode, defined according to the **adl_busAccess_t** structure.

  Length:

  Number of items to read from the bus.

  pDataToRead:

  Buffer where to copy the read items.

- Returned values
  - **OK** on success if the operation is pending (asynchronous mode).
  - A negative error value otherwise:
    - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
    - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value,
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

<u>Note:</u>

Items bit size is defined thanks to the **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command.

In asynchronous mode, the end of the read operation will be notified to the application through an interrupt event. Please refer to **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command for more information.

3.11.13 The adl_busReadExt Function

This function reads data from a previously subscribed bus SPI or I2C type.

<u>Note:</u>

By default the access is synchronous. This behavior can be changed with the **ADL_BUS_CMD_SET_ASYNC_MODE IOCtl** command.

- Prototype

```
s32   adl_busRead (s32              Handle,
                   adl_busAccess_t * pAccessMode,
                   u32              Length,
                   void *           pDataToRead
                   void *           context );
```

- Parameters

  Handle:

  Handle previously returned by the **adl_busSubscribe** function.

pAccessMode:

Bus access mode, defined according to the **adl_busAccess_t** structure.

Length:

Number of items to read from the bus.

pDataToRead:

Buffer where to copy the read items.

context:

Pointer on an application context, which will be provided back to the application when the asynchronous read operation end event will occur.

- Returned values
  - o **OK** on success
  - o A negative error value otherwise:
    - **Error** If a error during the operation occurs.**ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
    - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value,
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

Note:

Items bit size is defined thanks to the **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command.

In asynchronous mode, the end of the read operation will be notified to the application through an interrupt event. Please refer to **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command for more information.

3.11.14 The adl_busWrite Function

This function writes on a previously subscribed SPI or I2C bus type.

Note:

By default the access is synchronous. This behavior can be changed with the **ADL_BUS_CMD_SET_ASYNC_MODE IOCtl** command.

- Prototype

```
s32   adl_busWrite (s32                Handle,
                    adl_busAccess_t*   pAccessMode,
                    u32                Lenght,
                    void *             pDataToWrite );
```

- Parameters

Handle:

Handle previously returned by the **adl_busSubscribe** function.

pAccessMode:

Bus access mode, defined according to the **adl_busAccess_t structure**;

Length:

Number of items to write on the bus.

pDataToWrite:

Data buffer to write on the bus.

- Returned values
  - o **OK** on success if the operation is pending (asynchronous mode).
  - o A negative error value otherwise.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
    - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value,
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

Note:

Items bit size is defined thanks to the **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command.

In asynchronous mode, the end of the write operation will be notified to the application through an interrupt event. Please refer to **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command for more information.

3.11.15 The adl_busWriteExt Function

This function writes on a previously subscribed SPI or I2C bus type.

Note:

By default the access is synchronous. This behavior can be changed with the **ADL_BUS_CMD_SET_ASYNC_MODE IOCtl** command.

- Prototype

```
s32   adl_busWrite (s32                  Handle,
                    adl_busAccess_t*     pAccessMode,
                    u32                  Length,
                    void *               pDataToWrite
                     void *              context );
```

- Parameters

Handle:

Handle previously returned by the **adl_busSubscribe** function.

pAccessMode:

Bus access mode, defined according to the **adl_busAccess_t structure**;

Length:

Number of items to write on the bus.

pDataToWrite:

Data buffer to write on the bus.

context:

Pointer on an application context, which will be provided back to the application when the asynchronous read operation end event will occur.

- Returned values
  - o **OK** on success
  - o A negative error value otherwise.
    - **Error** If a error during the operation occurs,**ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
    - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value,
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler in synchronous mode (the function is forbidden in this context).

<u>Note:</u>

Items bit size is defined thanks to the **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command.

In asynchronous mode, the end of the write operation will be notified to the application through an interrupt event. Please refer to **ADL_BUS_CMD_SET_DATA_SIZE IOCtl** command for more information.

3.11.16 The adl_busDirectRead Function

This function reads data about previously subscribed Parallel bus type. This function is not usable with the SPI or I2C bus.

- Prototype

```
s32   adl_busDirectRead (s32             Handle,
                         u32             ChipAddress,
                         u32             DataLen,
                         void *          Data );
```

- Parameters

Handle:

Handle previously returned by the **adl_busSubscribe** function.

ChipAddress:

Chip address configuration. This address has to be a combination of the desired address bits to set. Available address bits are returned in a mask at subscription time.

DataLen:

Number of items to read from the bus.

Data:

Buffer into which the read items are copied, items bit size (8 or 16 bits) is defined at subscription time in the configuration structure (see **adl_busParallelSettings_t**).

- Returned values
  - **OK** on success
  - A negative error value otherwise.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
    - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value.

3.11.17 The adl_busDirectWrite Function

This function writes data on a previously subscribed Parallel bus type. This function is not usable with the SPI or I2C bus.

- Prototype

```
s32   adl_busDirectWrite (s32                  Handle,
                          u32                  ChipAddress,
                          u32                  Length,
                          void *               pDataToWrite );
```

- Parameters

Handle:

Handle previously returned by the **adl_busSubscribe** function.

ChipAddress:

Chip address configuration. This address has to be a combination of the desired address bits to set. Available address bits are returned in a mask at subscription time.

Length:

Number of items to write on the bus.

pDataToWrite:

Data buffer to write on the bus, item bit size (8 or 16 bits) is defined at subscription time in the configuration structure (see **adl_busParallelSettings_t**).

- Returned values
  - **OK** on success
  - A negative error value otherwise.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown,
    - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value.

3.11.18 Example

This example simply demonstrates how to use the BUS service in a nominal case (error cases are not handled) with a Wireless CPU®.

Complete examples of BUS service used are also available on the SDK.

```
// Global variables & constants

// SPI Subscription data
const adl_busSPISettings_t MySPIConfig =
{
    1,                          // No divider, use full clock speed
    ADL_BUS_SPI_CLK_MODE_0,     // Mode 0 clock
    ADL_BUS_SPI_ADDR_CS_GPIO,   // Use a GPIO to handle the Chip Select
                                   signal
    ADL_BUS_SPI_CS_POL_LOW,     // Chip Select active in low state
    ADL_BUS_SPI_MSB_FIRST,      // Data are sent MSB first
    ADL_IO_GPIO | 31,           // Use GPIO 31 to handle the Chip Select
                                   signal
    ADL_BUS_SPI_LOAD_UNUSED,    // LOAD signal not used
    ADL_BUS_SPI_DATA_BIDIR,     // 2 Wires configuration
    ADL_BUS_SPI_MASTER_MODE,    // Master mode
    ADL_BUS_SPI_BUSY_UNUSED     // BUSY signal not used
};

// I2C Subscription data
const adl_busI2CSettings_t MyI2CConfig =
{
    0x20,                       // Chip address is 0x20
    ADL_BUS_I2C_CLK_STD         // Chip uses the I2C standard clock speed
    ADL_BUS_I2C_ADDR_7_BITS,    // 7 bits address length
    ADL_BUS_I2C_MASTER_MODE     // Master mode
};

// Write/Read buffer sizes
#define WRITE_SIZE 5
#define READ_SIZE  3

// Access configuration structure
adl_busAccess_t AccessConfig =
{
    0, 0    // No Opcode, No Address
};
```

```
// BUS Handles
s32 MySPIHandle, MyI2Chandle;

// Data buffers
u8 WriteBuffer [ WRITE_SIZE ], ReadBuffer [ READ_SIZE ];

...

// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    s32 ReadValue;
    u32 AddSize=0;

    // Subscribe to the SPI1 BUS
    MySPIHandle = adl_busSubscribe ( ADL_BUS_ID_SPI, 1, &MySPIConfig );

    // Subscribe to the I2C BUS
    MyI2CHandle = adl_busSubscribe ( ADL_BUS_ID_I2C, 1, &MyI2CConfig );

    // Configure the Address length to 0 (rewrite the default value)
    adl_busIOCtl ( MySPIHandle, ADL_BUS_CMD_SET_ADD_SIZE, &AddSize );
    adl_busIOCtl ( MyI2CHandle, ADL_BUS_CMD_SET_ADD_SIZE, &AddSize );

    // Write 5 bytes set to '0' on the SPI & I2C bus
    wm_memset ( WriteBuffer, WRITE_SIZE, 0 );
    adl_busWrite ( MySPIHandle, &AccessConfig, WRITE_SIZE, WriteBuffer );
    adl_busWrite ( MyI2CHandle, &AccessConfig, WRITE_SIZE, WriteBuffer );

    // Read 3 bytes from the SPI & I2C bus
    adl_busRead ( MySPIHandle, &AccessConfig, READ_SIZE, ReadBuffer );
    adl_busRead ( MyI2CHandle, &AccessConfig, READ_SIZE, ReadBuffer );

    // Unsubscribe from subscribed BUS
    adl_busUnsubscribe ( MySPIHandle );
    adl_busUnsubscribe ( MyI2CHandle );
}
```

3.12 Error Management

ADL supplies Error service interface to allow the application to cause & intercept fatal errors, and also to retrieve stored back-trace logs. For the ADL standard error codes, please refer to section 0 Error Codes.

The defined operations are:

- A subscription function (**adl_errSubscribe**) to register an error event handler
- An unsubscription function (**adl_errUnsubscribe**) to cancel this event handler registration
- An error handler callback (**adl_errHdlr_f**) to be notified each time a fatal error occurs
- An error request function (**adl_errHalt**) to cause a fatal error
- A cleaning function (**adl_errEraseAllBacktraces**) to clean the back-traces storage area
- An analysis status function (**adl_errGetAnalysisState**) to retrieve the current back-trace analysis status
- An analysis start function (**adl_errStartBacktraceAnalysis**) to start the back-trace analysis
- A retrieve function (**adl_errRetrieveNextBacktrace**) to retrieve the next back-trace buffer for the current analysis.

3.12.1 Required Header File

The header file for the error functions is:

adl_error.h

3.12.2 Enumerations

3.12.2.1 The adl_ errInternalID_e Type

This type lists the error identifiers which should be generated by ADL.

- Code

```
typedef enum
{
          ADL_ERR_LEVEL_MEM = 0x0010,
          ADL_ERR_MEM_GET = ADL_ERR_LEVEL_MEM,
          ADL_ERR_MEM_RELEASE,
          ADL_ERR_LEVEL_FLH = 0x0020,
          ADL_ERR_FLH_READ = ADL_ERR_LEVEL_FLH,
          ADL_ERR_FLH_DELETE,
          ADL_ERR_LEVEL_APP = 0x0100
} adl_audioResources_e;
```

- Description

ADL_ERR_LEVEL_MEM:	Base level for generated ADL memory errors.
ADL_ERR_MEM_GET:	The platform runs out of dynamic memory.
ADL_ERR_MEM_RELEASE:	Internal error on dynamic memory release operation. Note: Internal usage only. An application has no way to produce such an error.
ADL_ERR_LEVEL_FLH:	Base level for generated ADL flash errors.
ADL_ERR_FLH_READ:	Internal error on flash object read operation. Note: Internal usage only. An application has no way to produce such an error
ADL_ERR_FLH_DELETE:	Internal error on flash object deletes operation. Note: Internal usage only. An application has no way to produce such an error
ADL_ERR_LEVEL_APP:	Base level for application generated errors.

3.12.2.2 The adl_errAnalysisState_e Type

This type is used to enumerate the possible states of the backtraces analysis.

- Code

```
typedef enum
{
        ADL_ERR_ANALYSIS_STATE_IDLE     // No running analysis
        ADL_ERR_ANALYSIS_STATE_RUNNING // A backtrace analysis is
                                          running
} adl_errAnalysisState_e;
```

3.12.3 Error event handler

Such a call-back is called each time a fatal error is caused by the application or by ADL.

Errors which should be generated by ADL are described in the **adl_errInternalID_e** type.

An error is described by an identifier and a string (associated text), that are sent as parameters to the **adl_errHalt** function.

If the error is processed and filtered the handler should return FALSE. The return value TRUE will cause the Wireless CPU® to execute a fatal error reset with a backtrace. A backtrace is composed of the provided message, and a call stack dump taken at the function call time. It is readable by the Target Monitoring Tool (Please refer to the Tools Manual [2] for more information).

- Prototype

```
typedef bool( * ) adl_errHdlr_f(u16 ErrorID, ascii *ErrorString)
```

- Parameters

  ErrorID

    Error identifier, defined by the application or by ADL

  ErrorString

    Error string, defined by the application or by ADL

- Returned values
  - TRUE If the handler decides to let the Wireless CPU® reset
  - FALSE If the handler refuses to let the Wireless CPU® reset

Note

An error event handler is called in the same execution context than the code which has caused the error.

If the error handler returns FALSE, the back-trace log is not registered in the Wireless CPU® non-volatile memory.

3.12.4 The adl_errSubscribe Function

This function subscribes to error service and gives an error handler: this allows the application to handle errors generated by ADL or by the **adl_errHalt** function. Errors generated by the Open AT® Firmware can not be handled by such an error handler.

- Prototype

```
s8    adl_errSubscribe (adl_errHdlr_f    ErrorHandler );
```

- Parameters

  ErrorHandler:

  Error Handler, Error event handler, defined using the **adl_errHdlr_f** type

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** if the parameter has an incorrect value
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service is already subscribed
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.12.5 The adl_errUnsubscribe Function

This function unsubscribes from error service. Errors generated by ADL or by the **adl_errHalt** function will no more are handled by the error handler.

- Prototype

```
s8    adl_errUnsubscribe     (adl_errHdlr_f    ErrorHandler);
```

- Parameters

  ErrorHandler:

  Error event handler, defined using the **adl_errHdlr_f** type, and previously provided to **adl_errSubscribe** function.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** if the parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handler is unknown
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if the service is not subscribed
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.12.6 The adl_errHalt Function

This function causes an error, defined by its ID and string. If an error handler is defined (using **adl_errHdlr_f** type), it will be called, otherwise a Wireless CPU® reset will occur.

When the Wireless CPU® resets (if there is no handler, or if this one returns TRUE), a back-trace log is registered in a non-volatile memory area, and also sent to the Target Monitoring Tool (if this one is running).

Such a back-trace log contains:

- the call stack dump when the error occurs
- the provided error identifier & string
- the context name which has caused the error, following the same behaviour than a trace display operation (please refer to the Debug Traces service for more information).

- Prototype

```
void   adl_errHalt (u16              ErrorID
                    const ascii *    ErrorStr );
```

- Parameters

ErrorID:

Error ID Error identifier. Shall be at least equal to **ADL_ERR_LEVEL_APP** (lower values are reserved for ADL internal error events)

ErrorStr:

Error string to be provided to the error handler, and to be stored in the resulting backtrace if a fatal error is required.

Note:

Please note that only the string address is stored in the backtrace, so this parameter has not to be a pointer on a RAM buffer, but a constant string pointer. Moreover, the string will only be correctly displayed if the current application is still present in the Wireless CPU®s flash memory. If the application is erased or modified, the string will not be correctly displayed when retrieving the backtraces.

Error identifiers below **ADL_ERR_LEVEL_APP** are for internal purpose so the application should only use an identifier above **ADL_ERR_LEVEL_APP**

When the Wireless CPU® reset is due to a fatal error, the init type parameter will be set to the **ADL_INIT_REBOOT_FROM_EXCEPTION** value (Please refer to the Tasks Initialization Service for more information).

3.12.7 The adl_errEraseAllBacktraces Function

Backtraces (caused by the **adl_errHalt** function, ADL or the Firmware) are stored in the Wireless CPU® non-volatile memory. A limited number of backtraces may be stored in memory (depending on each backtrace size, and other internal parameters stored in the same storage place). The **adl_errEraseAllBacktraces** function allows to free and re-initialize this storage place.

- Prototype

```
s32   adl_errEraseAllBacktraces ( void );
```

- Returned values
  - **OK** on success.**ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.12.8 The adl_errStartBacktraceAnalysis Function

In order to retrieve backtraces from the product memory, a backtrace analysis process has to be started with the **adl_errStartBacktraceAnalysis** function.

- Prototype

```
s8    adl_errStartBacktraceAnalysis ( void );
```

- Returned values
  - Handle A positive or null handle on success. This handle has to be used in the next **adl_errRetrieveNextBacktrace** function call. It will be valid until this function returns a **ADL_RET_ERR_DONE** code.
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if a backtrace analysis is already running.
  - **ERROR** if an unexpected internal error occurred.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

Note:

Only one analysis may be running at a time. The **adl_errStartBacktraceAnalysis** function will return the **ADL_RET_ERR_ALREADY_SUBSCRIBED** error code if it is called while an analysis is currently running.

3.12.9 The adl_errGetAnalysisState Function

This function may be used in order to know the current backtrace analysis process state.

- Prototype

```
adl_errAnalysisState_e  adl_errGetAnalysisState ( void );
```

- Returned values

  The current analysis state, using the **adl_errAnalysisState_e** type.

3.12.10 The adl_errRetrieveNextBacktrace Function

This function allows the application to retrieve the next backtrace buffer stored in the Wireless CPU® memory. The backtrace analysis has to be started first with the **adl_errStartBacktraceAnalysis** function.

- Prototype

```
s32   adl_errRetrieveNextBacktrace  (u8          Handle
                                     u8 *        BacktraceBuffer
                                     u16         Size );
```

- Parameters

Handle:

Backtrace analysis handle, returned by the **adl_errStartBacktraceAnalysis** function.

BacktraceBuffer:

Buffer in which the next retrieved backtrace will be copied. This parameter may be set to **NULL** in order to know the next backtrace buffer required size.

Size:

Backtrace buffer size. If this size is not large enough, the **ADL_RET_ERR_PARAM** error code will be returned.

- Returned values
  - o **OK** if the next stored backtrace was successfully copied in the BacktraceBuffer parameter.
  - o **Size**: the required size for next backtrace buffer if the BacktraceBuffer parameter is set to **NULL**.
  - o **ADL_RET_ERR_PARAM** if the provided Size parameter is not large enough.
  - o **ADL_RET_ERR_NOT_SUBSCRIBED** if the **adl_errStartBacktraceAnalysis** function was not called before.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the provided Handle parameter is invalid.
  - o **ADL_RET_ERR_DONE** if the last backtrace buffer has already been retrieved. The Handle parameter will now be unsubscribed and not usable any more with the **adl_errRetrieveNextBacktrace** function. A new analysis has to be started with the **adl_errStartBacktraceAnalysis** function.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

Note:

Once retrieved, the backtrace buffers may be stored (separately or concatenated), in order to be sent (using the application's protocol/bearer choice) to a remote server or PC. Once retrieved as one or several files on a PC, this (these) one(s) may be read using the Target Monitoring Tool and the Serial Link Manager in order to decode the backtrace buffer(s). Please refer to the Tools Manual (document [2]) in order to know how to process these files.

If **adl_errRetrieveNextBacktrace** is used you have to retrieve all next backtraces. Otherwise it is impossible to retrieve the first backtraces. There is no way to cancel a backtrace analysis; an analysis has always to be completed until all the backtraces are retrieved.

3.12.11 Example

The code sample below illustrates a nominal use case of the ADL Error service public interface (error cases are not handled).

```
// Error Event handler
bool MyErrorHandler ( u16 ErrorID, ascii * ErrorStr )
{
    // Nothing to do but accept the reset
    return TRUE;
}

// Error string
const ascii * MyErrorString = "Application Generated Error";

// Error launch function
void MyFunction1 ( void )
{
    // Subscribe to error service
    adl_errSubscribe ( MyErrorHandler );

    // Cause an error
    adl_errHalt ( ADL_ERR_LEVEL_APP + 1, MyErrorString );
}

// Error service unsubscription function
void MyFunction2 ( void )
{
    // Unsubscribe from error service
    adl_errUnsubscribe ( MyErrorHandler );
}

// Backtraces analysis event handler
u8 * MyAnalysisFunction ( void )
{
    // Start analysis
    s8 AnalysisHandle = adl_errStartBacktraceAnalysis();

    // Get state
    adl_errAnalysisState_e State = adl_errGetAnalysisState();

    // Retrieve next backtrace size
    u8 * Buffer = NULL;
    u32 Size = adl_errRetrieveNextBacktrace ( AnalysisHandle, Buffer, 0 );

    // Retrieve next backtrace buffer
    Buffer = adl_memGet ( Size );
    adl_errRetrieveNextBacktrace ( AnalysisHandle, Buffer, Size );

    // Erase all backtraces
    adl_errEraseAllBacktraces();

    // Return backtrace buffer
    return Buffer;
}
```

3.13 SIM Service

ADL provides this service to handle SIM and PIN code related events.

3.13.1 Required Header File

The header file for the SIM related functions is:

```
adl_sim.h
```

3.13.2 The adl_simSubscribe Function

This function subscribes to the SIM service, in order to receive SIM and PIN code related events. This will allow to enter PIN code (if provided) if necessary.

- Prototype

```
s32 adl_simSubscribe    ( adl_simHdlr_f    SimHandler,
                          ascii *          PinCode );
```

- Parameters

SimHandler:

SIM handler defined using the following type:

```
typedef void ( * adl_simHdlr_f ) ( u8 Event );
```

The events received by this handler are defined below.

<u>Normal events:</u>

ADL_SIM_EVENT_PIN_OK

if PIN code is all right

ADL_SIM_EVENT_REMOVED

if SIM card is removed

ADL_SIM_EVENT_INSERTED

if SIM card is inserted

ADL_SIM_EVENT_FULL_INIT

when initialization is done

<u>Error events:</u>

ADL_SIM_EVENT_PIN_ERROR

if given PIN code is wrong

ADL_SIM_EVENT_PIN_NO_ATTEMPT

if there is only one attempt left to entered the right PIN code

ADL_SIM_EVENT_PIN_WAIT

if the argument PinCode is set to NULL

<u>On the last three events, the service is waiting for the external application to enter the PIN code.</u>

Please note that the deprecated ADL_SIM_EVENT_ERROR event has been removed since the ADL version 3. This code was mentioned in version 2 documentation, but was never generated by the SIM service.

PinCode:

It is a string containing the PIN code text to enter. If it is set to NULL or if the provided code is incorrect, the PIN code will have to be entered by the external application.

This argument is used only the first time the service is subscribed. It is ignored on all further subscriptions.

- Returned value
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).
  - o **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service was already subscribed with the same handler.
  - o **ADL_RET_ERR_PARAM** if the function was called with a null handler.
  - o **OK** if the function is successfully executed.

3.13.3 The adl_simUnsubscribe Function

This function unsubscribes from SIM service. The provided handler will not receive SIM events any more.

- Prototype

```
s32   adl_simUnsubscribe     ( adl_simHdlr_f          Handler)
```

- Parameters

Handler:

Handler used with **adl_SimSubscribe** function.

- Returned value
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).
  - o **OK** if the function is successfully executed.

3.13.4 The adl_simGetState Function

This function gets the current SIM service state.

- Prototype

```
void  adl_simState_e adl_simGetState ( void );
```

- Returned values

The returned value is the SIM service state, based on following type:

```
typedef enum
{
  ADL_SIM_STATE_INIT,      // Service init state (PIN state not known yet)
  ADL_SIM_STATE_REMOVED,   // SIM removed
  ADL_SIM_STATE_INSERTED, // SIM inserted (PIN state not known yet)
  ADL_SIM_STATE_FULL_INIT, // SIM Full Init done
  ADL_SIM_STATE_PIN_ERROR, // SIM error state
  ADL_SIM_STATE_PIN_OK,    // PIN code OK, waiting for full init
  ADL_SIM_STATE_PIN_WAIT, // SIM inserted, PIN code not entered yet
  /* Always last State */
  ADL_SIM_STATE_LAST
} adl_simState_e;
```

3.13.5 adl_simEnterPIN Function

The **adl_simEnterPIN** interface enables the user to enter a new Pin Code.

- Prototype

```
s32 adl_simEnterPIN ( ascii * PinCode );
```

- Parameters

ascii * PinCode

a string holding the new Pin Code

- Returned values
  - **0** if the new Pin Code has been correctly processed
  - **ADL_RET_ERR_PARAM** if the Pin Code is not informed
  - **ADL_RET_ERR_BAD_STATE** if the SIM is not waiting for any Pin Code to be entered

Notes:

The Pin Code value is not definitively saved by the ADL SIM service and it is lost after each reset.

The ADL SIM service doesn't try to used the Pin Code provided if there is only one attempt left to entered the right PIN code.

3.14 Open SIM Access Service

The ADL Open SIM Access (OSA) service allows the application to handle APDU requests & responses with an external SIM card, connected through one of the Wireless CPU® interfaces (UART, SPI, I2C).

Note:

The Open SIM Access feature has to be enabled on the Wireless CPU® in order to make this service available.

The Open SIM Access feature state can be read thanks to the AT+WCFM=5 command response value: this feature state is represented by the bit 5 (00000020 in hexadecimal format).

Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU®.

3.14.1 Required Header File

The header file for the OSA service definitions is:

```
adl_osa.h
```

3.14.2 The adl_osaSubscribe Function

This function allows the application to supply an OSA service handler, which will then be notified on each OSA event reception.

Moreover, by calling this function, the application requests the Wavecom firmware to close the local SIM connection, and to post SIM requests to the application from now.

- Prototype

```
s32 adl_osaSubscribe ( adl_osaHandler_f OsaHandler );
```

- Parameters

  OsaHandler:

  OSA service handler supplied by the application.

  Please refer to **adl_osaHandler_f** type definition for more information (see paragraph 3.14.3).

- Returned values
  - A positive or null value on success:
    OSA service handle, to be used in further OSA service function calls. A confirmation event will then be received in the service handler:
    - **ADL_OSA_EVENT_INIT_SUCCESS** if the local SIM connection was closed successfully,
    - **ADL_OSA_EVENT_INIT_FAILURE** if a Bluetooth SAP connection is running.
  - A negative **error** value otherwise:
    - **ADL_RET_ERR_PARAM** on a supplied parameter error,

- **ADL_RET_ERR_NOT_SUPPORTED** if the Open SIM access feature is not enabled on the Wireless CPU®
- **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service was already subscribed (the OSA service can only be subscribed one time).
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.14.3 The adl_osaHandler_f call-back Type

Such a call-back function has to be supplied to ADL on the OSA service subscription. It will be notified by the service on each OSA event.

- Prototype

```
typedef void (* adl_osaHandler_f) ( adl_osaEvent_e          Event,
                                    adl_osaEventParam_u *   Param );
```

- Parameters

Event:

OSA service event identifier, using one of the following defined values.

Event Type	Use
ADL_OSA_EVENT_INIT_SUCCESS	The OSA service has been successfully subscribed: The local SIM card has been shut down, and, From now on, all SIM requests will be posted to on the application through the OSA service.
ADL_OSA_EVENT_INIT_FAILURE	The OSA service subscription has failed: The Wireless CPU® is already connected to a remote SIM through the Bluetooth SAP profile (the SAP connection has to be closed prior to subscribing to the OSA service).
ADL_OSA_EVENT_ATR_REQUEST	The application is notified with this event after the ADL_OSA_EVENT_INIT_SUCCESS one: The Wavecom firmware is required for the Answer To Reset data. The application has to reset the remote SIM card, and to get the ATR data in order to post it back to the Wavecom firmware through the **adl_osaSendResponse** function.

Event Type	Use
ADL_OSA_EVENT_APDU_REQUEST	This event is received by the application each time the Wavecom firmware has to send an APDU request to the SIM card. This request (notified to the application through the **Length** & **Data** parameters) has to be forwarded to the remote SIM by the application, and has to read the associated response in order to post it back to the Wavecom firmware through the **adl_osaSendResponse** function.
ADL_OSA_EVENT_SIM_ERROR	This event is notified to the application: If an error was notified to the Wavecom firmware in a SIM response (posted through the **adl_osaSendResponse** function), or, If the internal response time-out has elapsed (a request event was sent to the application, but no response was posted back to the Wavecom firmware). When this event is received, the OSA service is automatically un-subscribed and the Wavecom firmware resumes the local SIM connection.
ADL_OSA_EVENT_CLOSED	The application will receive this event after un-subscribing from the OSA service. The Wavecom firmware has resumed the local SIM connection.

Param

Event parameters, using the following type:

```
typedef union
{
       adl_osaStatus_e   ErrorEvent;
       struct {
       {
           u16 Length;
           u8 * Data;
       }                 RequestEvent;
} adl_osaEventParam_u;
```

This union is used depending on the event type.

Event Type	Event Parameter
ADL_OSA_EVENT_INIT_SUCCESS	Set to NULL
ADL_OSA_EVENT_INIT_FAILURE	Set to NULL
ADL_OSA_EVENT_ATR_REQUEST	Set to NULL
ADL_OSA_EVENT_APDU_REQUEST	**RequestEvent** structure set: **Length:** APDU request buffer length **Data:** APDU request data buffer address
ADL_OSA_EVENT_SIM_ERROR	**ErrorEvent** value set, according to the status previously sent back through the **adl_osaSendResponse** function, or set by the firmware on unsolicited errors. Please refer to the **adl_osaSendResponse** function description for more information.
ADL_OSA_EVENT_CLOSED	Set to NULL

3.14.4 The adl_osaSendResponse Function

This function allows the application to post back ATR or APDU responses to the Wavecom firmware, after receiving an **ADL_OSA_EVENT_ATR_REQUEST** or **ADL_OSA_EVENT_APDU_REQUEST** event.

- Prototype

```
s32 adl_osaSendResponse (s32              OsaHandle,
                         adl_osaStatus_e  Status,
                         u16              Length,
                         u8 *             Data );
```

- Parameters

OsaHandle:

OSA service handle, previously returned by the **adl_osaSubscribe** function.

Status

Status to be supplied to the firmware, in response to an ATR or APDU request, using the following defined values.

Event Type	Use
ADL_OSA_STATUS_OK	Response data buffer has been received from the SIM card.
ADL_OSA_STATUS_CARD_NOT_ACCESSIBLE	SIM card does not seem to be accessible (no response from the card).
ADL_OSA_STATUS_CARD_REMOVED	The SIM card has been removed.
ADL_OSA_STATUS_CARD_UNKNOWN_ERROR	Generic code for all other error cases.

Length:

ATR or APDU request response buffer length, in bytes.

Note:

Should be set to 0 if the SIM card status is not OK.

Data:

ATR or APDU request response buffer address. This buffer content will be copied and sent by ADL to the Wavecom firmware.

Note:

Should be set to 0 if the SIM card status is not OK.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** on a supplied parameter error.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied OSA handle is unknown.
  - **ADL_RET_ERR_BAD_STATE** if the OSA service is not waiting for an APDU or ATR request response.

3.14.5 The adl_osaUnsubscribe Function

This function un-subscribes from the OSA service: the local SIM connection is resumed by the Wavecom Firmware, and the application supplied handler is not any longer notified of OSA events.

- Prototype

```
s32 adl_osaUnsubscribe ( s32 OsaHandle );
```

- Parameters

OsaHandle:

OSA service handle, previously returned by the **adl_osaSubscribe** function.

- Returned values
  - **OK** on success.

    An **ADL_OSA_EVENT_CLOSED** confirmation event will then be received in the service handler.

- o **ADL_RET_ERR_UNKNOWN_HDL** if the supplied OSA handle is unknown.
- o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).
- o **ADL_RET_ERR_NOT_SUBSCRIBED** The OSA service is not subscribed, so it is not possible to unsubscribe it.
- o **ADL_RET_ERR_BAD_STATE** Firmware is waiting for an ATR or APDU request from the simcard, and unsubscription is forbidden until the simcard's request is granted.

3.14.6 Example

This example simply demonstrates how to use the OSA service in a nominal case (error cases are not handled).

```
// Global variables


// OSA service handle
s32 OsaHandle;

// SIM request response data buffer length & address
u16 SimRspLen;
u8 * SimRspData;


  // OSA service handler
void MyOsaHandler ( adl_osaEvent_e Event, adl_osaEventParam_u * Param )
{
    // Switch on the event type
    switch ( Event )
    {
        case ADL_OSA_EVENT_ATR_REQUEST :
        case ADL_OSA_EVENT_APDU_REQUEST :
            // Reset the SIM card or transmit request
            // Get the related response data buffer
            // To be copied to SimRspLen & SimRspData global variables
            // Post back the response to the Wavecom firmware
            adl_osaSendResponse  (     OsaHandle,ADL_OSA_STATUS_OK,
   SimRspLen, SimRspData );
        break;
    }
}
// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribes to the OSA service
    OsaHandle = adl_osaSubscribe ( MyOsaHandler );
}
void MyFunction2 ( void )
{
    // Un-subscribes from the OSA service
    adl_osaUnsubscribe ( OsaHandle );
}
```

3.15 SMS Service

ADL provides this service to handle SMS events, and to send SMSs to the network.

3.15.1 Required Header File

The header file for the SMS related functions is:

```
adl_sms.h
```

3.15.2 The adl_smsSubscribe Function

This function subscribes to the SMS service in order to receive SMSs from the network.

- Prototype

```
s8    adl_smsSubscribe (adl_smsHdlr_f          SmsHandler,
                         adl_smsCtrlHdlr_f      SmsCtrlHandler,
                         u8                     Mode );
```

- Parameters

SmsHandler:

SMS handler defined using the following type:

```
typedef bool ( * adl_smsHdlr_f ) (ascii * SmsTel,
                                  ascii * SmsTimeLength,
                                  ascii * SmsText );
```

This handler is called each time an SMS is received from the network.

SmsTel contains the originating telephone number of the SMS (in text mode), or NULL (in PDU mode).

SmsTimeLength contains the SMS time stamp (in text mode), or the PDU length (in PDU mode).

SmsText contains the SMS text (in text mode), or the SMS PDU (in PDU mode).

This handler returns TRUE if the SMS must be forwarded to the external application (it is then stored in SIM memory, and the external application is then notified by a "+CMTI" unsolicited indication).

It returns FALSE if the SMS should not be forwarded.

If the SMS service is subscribed several times, a received SMS will be forwarded to the external application only if each of the handlers return TRUE.

Note:

Whatever is the handler's returned value, the incoming message has been internally processed by ADL; if it is read later via the +CMGR or +CMGL command, its status will be 'REC READ', instead of 'REC UNREAD'.

SmsCtrlHandler:

SMS event handler, defined using the following type:

```
typedef void ( * adl_smsCtrlHdlr_f ) (u8   Event,
                                      u16  Nb );
```

This handler is notified by following events during a n sending process.

ADL_SMS_EVENT_SENDING_OK

*the SMS was sent successfully, **Nb** parameter value is not relevant.*

ADL_SMS_EVENT_SENDING_ERROR

*An error occurred during SMS sending, **Nb** parameter contains the error number, according to "+CMS ERROR" value (cf. AT Commands Interface Guide).*

ADL_SMS_EVENT_SENDING_MR

the SMS was sent successfully, Nb parameter contains the sent Message Reference value. A ADL_SMS_EVENT_SENDING_OK event will be received by the control handler.

Mode:

Mode used to receive SMSs:

ADL_SMS_MODE_PDU

SmsHandler will be called in PDU mode on each SMS reception.

ADL_SMS_MODE_TEXT

SmsHandler will be called in Text mode on each SMS reception.

- Returned values
  - o On success, this function returns a positive or null handle, requested for further SMS sending operations.
  - o **ADL_RET_ERR_PARAM** if a parameter has a wrong value.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.15.3 The adl_smsSend Function

This function sends an SMS to the network.

- Prototype

```
s8    adl_smsSend (u8         Handle,
                   ascii *    SmsTel,
                   ascii *    SmsText,
                   u8         Mode );
```

- Parameters

Handle:

Handle returned by **adl_smsSubscribe** function.

SmsTel:

Telephone number where to send the SMS (in text mode), or NULL (in PDU mode).

SmsText:

SMS text (in text mode), or SMS PDU (in PDU mode).

Mode:

Mode used to send SMSs:

ADL_SMS_MODE_PDU

to send a SMS in PDU mode.

ADL_SMS_MODE_TEXT

to send a SMS in Text mode.

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_PARAM** if a parameter has an incorrect value.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - o **ADL_RET_ERR_BAD_STATE** if the product is not ready to send an SMS (initialization not yet performed, or sending an SMS already in progress)
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.15.4 The adl_smsUnsubscribe Function

This function unsubscribes from the SMS service. The associated handler with provided handle will no longer receive SMS events.

- Prototype

```
s8    adl_smsUnsubscribe     ( u8  Handle)
```

- Parameters

Handle:

Handle returned by **adl_smsSubscribe** function.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handler is unknown.
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if the service is not subscribed.
  - **ADL_RET_ERR_BAD_STATE** if the service is processing an SMS
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.16 Message Service

ADL provides this service to allow applications to post and handle messages. Messages are used to exchange data between the different application components (application task, Interrupt handler...).

The defined operations are:

- subscription & unsubscription functions (**adl_msgSubscribe** & **adl_msgUnsubscribe**) usable to manage message reception filters.
- reception callbacks (**adl_msgHandler_f**) usable to receive incoming messages.
- A sending function (**adl_msgSend**) usable to send messages to an application task.

3.16.1 Required Header File

The header file for message-related functions is:

```
adl_msg.h
```

3.16.2 The adl_msgIdComparator_e Type

Enumeration of comparison operators, usable to define a message filter through the adl_msgFilter_t structure..

```
typedef enum
{
  ADL_MSG_ID_COMP_EQUAL,
  ADL_MSG_ID_COMP_DIFFERENT,
  ADL_MSG_ID_COMP_GREATER,
  ADL_MSG_ID_COMP_GREATER_OR_EQUAL,
  ADL_MSG_ID_COMP_LOWER,
  ADL_MSG_ID_COMP_LOWER_OR_EQUAL,
  ADL_MSG_ID_COMP_LAST,              //Reserved for internal use
} adl_msgIdComparator_e;
```

The meaning of each comparison operator is defined below:

Comparison Operator	Description
`ADL_MSG_ID_COMP_EQUAL`	The two identifiers are equal.
`ADL_ MSG_ID_COMP_DIFFERENT`	The two identifiers are different.
`ADL_ MSG_ID_COMP_GREATER`	The received message identifier is greater than the subscribed message identifier.
`ADL_ MSG_ID_COMP_GREATER_OR_EQUAL`	The received message identifier is greater or equal to the subscribed message identifier.

Comparison Operator	Description
ADL_ MSG_ID_COMP_LOWER	The received message identifier is lower than the subscribed message identifier.
ADL_ MSG_ID_COMP_LOWER_OR_EQUAL	The received message identifier is lower or equal to the subscribed message identifier.

3.16.3 The adl_msgFilter_t Structure

This structure allows the application to define a message filter at service subscription time.

```
typedef struct
{
  u32                         MsgIdentifierMask;
  u32                         MsgIdentifierValue;
  adl__msgIdComparator_e      Comparator;
  adl__ctxID_e                Source;
} adl_msgFilter_t;
```

3.16.3.1 Structure Fields

The structure fields are defined below:

- **MsgIdentifierMask**:
  Bit mask to be applied to the incoming message identifier at reception time. Only the bits set to 1 in this mask will be compared for the service handlers notification. If the mask is set to 0, the identifier comparison will always match.
  MsgIdentifierValue:
  Message identifier value to be compared with the received message identifier. Only the bits filtered by the **MsgIdentifierMask** mask are significant.
- **Comparator**:
  Operator to be used for incoming message identifier comparison, using the **adl_msgIdComparator_e** type. Please refer to the type description for more information (see § 3.16.2).
- **Source**:
  Required incoming message source context: the handler will be notified with messages received from this context. The **ADL_CTX_ALL** constant should be used if the application wishes to receive all messages, whatever the source context.

3.16.3.2 Filter Examples

- With the following filter parameters:
  `MsgIdentifierMask = 0x0000F000`
  `MsgIdentifierValue = 0x00003000`
  `Comparator = ADL_MSG_ID_COMP_EQUAL`
  `Source = ADL_CTX_ALL`
  the comparison will match if the message identifier fourth quartet is strictly equal to 3, whatever the other bit values, and whatever the source context.

- With the following filter parameters:
  `MsgIdentifierMask = 0`
  `MsgIdentifierValue = 0`
  `Comparator = ADL_MSG_ID_COMP_EQUAL`
  `Source = ADL_CTX_ALL`
  the comparison will always match, whatever the message identifier & the source context values

- With the following filter parameters:
  `MsgIdentifierMask = 0xFFFF0000`
  `MsgIdentifierValue = 0x00010000`
  `Comparator = ADL_MSG_ID_COMP_GREATER_OR_EQUAL`
  `Source = ADL_CTX_HIGH_LEVEL_IRQ_HANDLER`
  the comparison will match if the message identifier two most significant bytes are greater or equal to 1, and if the message was posted from high level Interrupt handler.

3.16.4 The adl_msgSubscribe Function

This function allows the application to receive incoming user-defined messages, sent from any application components (the application task itself or Interrupt handlers).

- Prototype

```
s32   adl_msgSubscribe  (adl_mgsFilter_t_ *      Filter,
                         adl_msgHandler_f        msgHandler);
```

- Parameters

  Filter:

  Identifier and source context conditions to check each message reception in order to notify the message handler. Please refer to the **`adl_msgFilter_t`** structure description for more information.

  MsgHandler:

  Application defined message handler, which will be notified each time a received message matches the filter conditions. Please refer to **`adl_msgHandler_f`** call-back type definition for more information.

- Returned values
  - A positive or null value on success:

        - Message service handle, to be used in further Message service functions calls.
    - A negative error value otherwise:
        - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value.
        - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

Note:

Messages filters definition is specific to each task: the filter will apply only to incoming messages for the current task context. The associated call-back will be called in this task context when the filter conditions are fulfilled.

3.16.5 The adl_msgHandler_f call-back Type

Such a call-back function has to be supplied to ADL through the **adl_msgSubscribe** interface in order to receive incoming messages. Messages will be received through this handler each time the supplied filter conditions are fulfilled.

- Prototype

```
typedef void (*adl_msgHandler_f) ( u32          MsgIdentifier,
                                   adl_ctxID_e  Source,
                                   u32          Length,
                                   void *       Data );
```

- Parameters

MsgIdentifier:

Incoming message identifier.

Source:

Source context identifier from which the message was sent.

Length:

Message body length, in bytes. This length should be 0 if the message does not include a body.

Data:

Message body buffer address. This address should be NULL if the message does not include a body.

Note:

A message handler callback will be called by ADL in the execution context where it has been subscribed.

3.16.6 The adl_msgUnsubscribe Function

This function un-subscribes from a previously subscribed message filter. Associated message handler will no longer receive the filtered messages.

- Prototype

```
S32   adl_msgUnsubscribe    (s32    MsgHandle );
```

- Parameters

  MsgHandle:

  Handle previously returned by the **adl_msgSubscribe** function.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied handle is unknown.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.16.7 The adl_msgSend Function

This function allows the application to send a message at any time to any running task.

- Prototype

```
s32   adl_msgSend (adl_ctxID_e      DestinationTask,
                   u32              MessageIdentifier,
                   u32              Length,
                   void *           Data );
```

- Parameters

  DestinationTask:

  Destination task to which the message is to be posted, using the adl_ctxID_e type. Only tasks identifiers are valid (it is not possible to post messages to interrupt handler contexts).

  MessageIdentifier:

  The application defined message identifier. Message reception filters will be applied to this identifier before notifying the concerned message handlers.

  Length:

  Message body length, if any. Should be set to 0 if the message does not include a body.

  Data:

  Message body buffer address, if any. Should be set to 0 if the message does not include a body. This buffer data content will be copied into the message.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** if a parameter has an incorrect value.

Note:

When a message is posted, the source context identifier is automatically set accordingly to the current context:

- If the message is sent from the application task, the source context identifier is set to **ADL_CTX_OAT_TASK**.

- If the message is sent from a low level Interrupt handler, the source context identifier is set to **ADL_CTX_LOW_LEVEL_IRQ_HANDLER**.
- If the message is sent from a high level Interrupt handler, the source context identifier is set to **ADL_CTX_HIGH_LEVEL_IRQ_HANDLER**.

3.16.8 Example

The code sample below illustrates a nominal use case of the ADL Messages Service public interface (error cases are not handled).

```
// Global variables & constants

// Message filter definition
const adl_msgFilter_t MyFilter =
{
    0xFFFF0000,                         // Compare only the 2 MSB
    0x00010000,                         // Compare with 1
    ADL_MSG_ID_COMP_GREATER_OR_EQUAL,   // Msg ID has to be >= 1
    0                                   // Application task 0 incoming msg
                                           only
};

// Message service handle
s32 MyMsgHandle;

// Incoming message handler
void MyMsgHandler ( u32 MsgIdentifier, adl_ctxID_e Source, u32 Length, void
* Data )
{
  // Message processing
}

// Somewhere in the application code
void MyFunction ( void )
{
    // Subscribe to the message service
    MyMsgHandle = adl_msgSubscribe ( &MyFilter, MyMsgHandler );

    // Send an empty message to task 0
    adl_msgSend ( 0, 0x00010055, 0, NULL );

    // Unsubscribe from the message service
    adl_msgUnsubscribe ( MyMsgHandle );
}
```

3.17 Call Service

ADL provides this service to handle call related events, and to setup calls.

3.17.1 Required Header File

The header file for the call related functions is:

```
adl_call.h
```

3.17.2 The adl_callSubscribe Function

This function subscribes to the call service in order to receive call related events.

- Prototype

```
s8    adl_callSubscribe ( adl_callHdlr_f  CallHandler );
```

- Parameters

CallHandler:

Call handler defined using the following type:

```
typedef s8 ( * adl_callHdlr_f ) ( u16    Event,
                                  u32    Call_ID );
```

The pair events / call Id received by this handler are defined below; each event is received according to an "event type", which can be:

- MO (Mobile Originated call related event)
- MT (Mobile Terminated call related event)
- CMD (Incoming AT command related event)

Event / Call ID	Description	Type
ADL_CALL_EVENT_RING_VOICE / 0	if voice phone call	MT
ADL_CALL_EVENT_RING_DATA / 0	if data phone call	MT
ADL_CALL_EVENT_NEW_ID / X	if wind: 5,X	MO MT [1]
ADL_CALL_EVENT_RELEASE_ID / X	if wind: 6,X ; on data call release, X is a logical OR between the Call ID and the ADL_CALL_DATA_FLAG constant	MO MT
ADL_CALL_EVENT_ALERTING / 0	if wind: 2	MO

[1] In case of Call Waiting only; please refer to the AT Commands Interface Guide [1] for more information.

Event / Call ID	Description	Type
ADL_CALL_EVENT_NO_CARRIER / 0	phone call failure, 'NO CARRIER'	MO MT
ADL_CALL_EVENT_NO_ANSWER / 0	phone call failure, no answer	MO
ADL_CALL_EVENT_BUSY / 0	phone call failure, busy	MO
ADL_CALL_EVENT_SETUP_OK / Speed	**OK** response after a call setup performed by the **adl_callSetup** function; in data call setup case, the connection <Speed> (in bits/second) is also provided.	MO
ADL_CALL_EVENT_ANSWER_OK / Speed	**OK** response after an **ADL_CALL_NO_FORWARD_ATA** request from a call handler ; in data call answer case, the connection <Speed> (in bps) is also provided	MT
ADL_CALL_EVENT_CIEV / Speed	**OK** response after a performed call setup; in data call setup case, the connection <Speed> (in bps) is also provided	
ADL_CALL_EVENT_HANGUP_OK / Data	**OK** response after a **ADL_CALL_NO_FORWARD_ATH** request, or a call hangup performed by the adl_callHangup function ; on data call release, Data is the **ADL_CALL_DATA_FLAG** constant (0 on voice call release)	MO MT
ADL_CALL_EVENT_SETUP_OK_FROM_EXT / Speed	**OK** response after an 'ATD' command from the external application; in data call setup case, the connection <Speed> (in bits/second) is also provided.	MO
ADL_CALL_EVENT_ANSWER_OK_FROM_EXT / Speed	**OK** response after an 'ata' command from the external application ; in data call answer case, the connection <Speed> (in bps) is also provided	MT
ADL_CALL_EVENT_HANGUP_OK_FROM_EXT / Data	**OK** response after an 'ATH' command from the external application ; on data call release, Data is the **ADL_CALL_DATA_FLAG** constant (0 on voice call release)	MO MT
ADL_CALL_EVENT_AUDIO_OPENNED / 0	if +WIND: 9	MO MT

Event / Call ID	Description	Type
ADL_CALL_EVENT_ANSWER_OK_AUTO / Speed	**OK** response after an auto-answer to an incoming call (ATS0 command was set to a non-zero value) ; in data call answer case, the connection <Speed> (in bps) is also provided	MT
ADL_CALL_EVENT_RING_GPRS / 0	if GPRS phone call	MT
ADL_CALL_EVENT_SETUP_FROM_EXT / Mode	if the external application has used the 'ATD' command to setup a call. Mode value depends on call type (Voice: 0, GSM Data: **ADL_CALL_DATA_FLAG**, GPRS session activation: binary OR between **ADL_CALL_GPRS_FLAG** constant and the activated CID). According to the notified handlers return values, the call setup may be launched or not: if at least one handler returns the **ADL_CALL_NO_FORWARD** code (or higher), the command will reply "+CME ERROR: 600" to the external application; otherwise (if all handlers return **ADL_CALL_FORWARD)** the call setup is launched.	CMD
ADL_CALL_EVENT_SETUP_ERROR_NO_SIM / 0	A call setup (from embedded or external application) has failed (no SIM card inserted)	MO
ADL_CALL_EVENT_SETUP_ERROR_PIN_NOT_READY / 0	A call setup (from embedded or external application) has failed (the PIN code is not entered)	MO
ADL_CALL_EVENT_SETUP_ERROR / Error	A call setup (from embedded or external application) has failed (the <Error> field is the returned +CME **ERROR** value ; cf. AT Commands interface guide for more information)	MO

The events returned by this handler are defined below:

Event	Description
ADL_CALL_FORWARD	the call event shall be sent to the external application On unsolicited events, these ones will be forwarded to all opened ports. On responses events, these ones will be forwarded only on the port on which the request was executed.
ADL_CALL_NO_FORWARD	the call event shall not be sent to the external application
ADL_CALL_NO_FORWARD_ATH	the call event shall not be sent to the external application and the application shall terminate the call by sending an 'ATH' command.
ADL_CALL_NO_FORWARD_ATA	the call event shall not be sent to the external application and the application shall answer the call by sending an 'ATA' command.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** on parameter error
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.17.3 The adl_callSetup Function

This function just runs the **adl_callSetupExt** one on the **ADL_PORT_OPEN_AT_VIRTUAL_BASE** port (cf. **adl_callSetupExt** description for more information). Please note that events generated by the **adl_callSetup** will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.17.4 The adl_callSetupExt Function

This function sets up a call to a specified phone number.

- Prototype

```
s8    adl_callSetupExt  (ascii *       PhoneNb,
                          u8            Mode,
                          adl_port_e    Port );
```

- Parameters

PhoneNb:

Phone number to use to set up the call.

Mode:

Mode used to set up the call:

ADL_CALL_MODE_VOICE,
ADL_CALL_MODE_DATA

Port:

Port on which to run the call setup command. When setup return events will be received in the Call event handler, if the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** on parameter error (bad value, or unavailable port)
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.17.5 The adl_callHangup Function

This function just runs the `adl_callHangupExt` one on the **ADL_PORT_OPEN_AT_VIRTUAL_BASE** port (cf. `adl_callHangupExt` description for more information). Please note that events generated by the `adl_callHangup` will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.17.6 The adl_callHangupExt Function

This function hangs up the phone call.

- Prototype

```
s8    adl_callHangupExt ( adl_port_e Port );
```

- Parameters

Port:

Port on which to run the call hang-up command. When hang-up return events will be received in the Call event handler, if the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** on parameter error (unavailable port)
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.17.7 The adl_callAnswer Function

This function just runs the **adl_callAnswerExt** one on the **ADL_PORT_OPEN_AT_VIRTUAL_BASE** port (cf. **adl_callAnswerExt** description for more information). Please note that events generated by the **adl_callAnswer** will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.17.8 The adl_callAnswerExt Function

This function allows the application to answer a phone call out of the call events handler.

- Prototype

```
s8    adl_callAnswerExt ( adl_port_e Port );
```

- Parameters

  Port:

  Port on which to run the call hang-up command. When hang-up return events will be received in the Call event handler, if the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** on parameter error (unavailable port)
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.17.9 The adl_callUnsubscribe Function

This function unsubscribes from the Call service. The provided handler will not receive Call events any more.

- Prototype

```
s8    adl_callUnsubscribe    ( adl_callHdlr_f Handler );
```

- Parameters

  Handler:

  Handler used with **adl_callSubscribe** function.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** on parameter error
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handler is unknown
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if the service is not subscribed.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.18 GPRS Service

ADL provides this service to handle GPRS related events and to setup, activate and deactivate PDP contexts.

3.18.1 Required Header File

The header file for the GPRS related functions is:

```
adl_gprs.h
```

3.18.2 The adl_gprsSubscribe Function

This function subscribes to the GPRS service in order to receive GPRS related events.

- Prototype

```
s8    adl_gprsSubscribe (adl_gprsHdlr_f    GprsHandler );
```

- Parameters

GprsHandler:

GPRS handler defined using the following type:

```
typedef s8 (*adl_gprsHdlr_f)(u16 Event, u8 Cid);
```

The pairs events/Cid received by this handler are defined below:

Event / Call ID	Description
ADL_GPRS_EVENT_RING_GPRS	If incoming PDP context activation is requested by the network
ADL_GPRS_EVENT_NW_CONTEXT_DEACT / X	If the network has forced the deactivation of the Cid X
ADL_GPRS_EVENT_ME_CONTEXT_DEACT / X	If the ME has forced the deactivation of the Cid X
ADL_GPRS_EVENT_NW_DETACH	If the network has forced the detachment of the ME
ADL_GPRS_EVENT_ME_DETACH	If the ME has forced a network detachment or lost the network
ADL_GPRS_EVENT_NW_CLASS_B	If the network has forced the ME on class B
ADL_GPRS_EVENT_NW_CLASS_CG	If the network has forced the ME on class CG
ADL_GPRS_EVENT_NW_CLASS_CC	If the network has forced the ME on class CC
ADL_GPRS_EVENT_ME_CLASS_B	If the ME has changed to class B
ADL_GPRS_EVENT_ME_CLASS_CG	If the ME has changed to class CG
ADL_GPRS_EVENT_ME_CLASS_CC	If the ME has changed to class CC

Event / Call ID	Description
ADL_GPRS_EVENT_NO_CARRIER	If the activation of the external application with 'ATD*99' (PPP dialing) did hang up.
ADL_GPRS_EVENT_DEACTIVATE_OK / X	If the deactivation requested with **adl_gprsDeact** function was successful on the Cid X
ADL_GPRS_EVENT_DEACTIVATE_OK_FROM_EXT / X	If the deactivation requested by the external application was successful on the Cid X
ADL_GPRS_EVENT_ANSWER_OK	If the acceptance of the incoming PDP activation with **adl_gprsAct** was successful
ADL_GPRS_EVENT_ANSWER_OK_FROM_EXT	If the acceptance of the incoming PDP activation by the external application was successful
ADL_GPRS_EVENT_ACTIVATE_OK / X	If the activation requested with **adl_gprsAct** on the Cid X was successful
ADL_GPRS_EVENT_GPRS_DIAL_OK_FROM_EXT / X	If the activation requested by the external application with 'ATD*99' (PPP dialing) was successful on the Cid X
ADL_GPRS_EVENT_ACTIVATE_OK_FROM_EXT / X	If the activation requested by the external application on the Cid X was successful
ADL_GPRS_EVENT_HANGUP_OK_FROM_EXT	If the rejection of the incoming PDP activation by the external application was successful
ADL_GPRS_EVENT_DEACTIVATE_KO / X	If the deactivation requested with **adl_gprsDeact** on the Cid X failed
ADL_GPRS_EVENT_DEACTIVATE_KO_FROM_EXT / X	If the deactivation requested by the external application on the Cid X failed
ADL_GPRS_EVENT_ACTIVATE_KO_FROM_EXT / X	If the activation requested by the external application on the Cid X failed
ADL_GPRS_EVENT_ACTIVATE_KO / X	If the activation requested with **adl_gprsAct** on the Cid X failed
ADL_GPRS_EVENT_ANSWER_OK_AUTO	If the incoming PDP context activation was automatically accepted by the ME
ADL_GPRS_EVENT_SETUP_OK / X	If the set up of the Cid X with **adl_gprsSetup** was successful
ADL_GPRS_EVENT_SETUP_KO / X	If the set up of the Cid X with **adl_gprsSetup** failed

Event / Call ID	Description
ADL_GPRS_EVENT_ME_ATTACH	If the ME has forced a network attachment
ADL_GPRS_EVENT_ME_UNREG	If the ME is not registered
ADL_GPRS_EVENT_ME_UNREG_SEARCHING	If the ME is not registered but is searching a new operator for registration.

Note:

If Cid X is not defined, the value **ADL_CID_NOT_EXIST** will be used as X.

The possible returned values for this handler are defined below:

Event	Description
ADL_GPRS_FORWARD	the event shall be sent to the external application. On unsolicited events, these one be forwarded to all opened ports. On responses events, these one be forwarded only on the port on which the request was executed.
ADL_GPRS_NO_FORWARD	the event is not sent to the external application
ADL_GPRS_NO_FORWARD_ATH	the event is not sent to the external application and the application will terminate the incoming activation request by sending an 'ATH' command.
ADL_GPRS_NO_FORWARD_ATA	the event is not sent to the external application and the application will accept the incoming activation request by sending an 'ATA' command.

- Returned values for adl_gprsSubscribe

This function returns **OK** on success, or a negative error value.

Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	In case of parameter error
ADL_RET_ERR_SERVICE_LOCKED	If the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.18.3 The adl_gprsSetup Function

This function runs the `adl_gprsSetupExt` on the ADL_PORT_OPEN_AT_VIRTUAL_BASE port (cf. `adl_gprsSetupExt` description for more information). Please note that events generated by the `adl_gprsSetup` will not be able to be forwarded to any external port, since the setup command runs on the Open AT® port.

3.18.4 The adl_gprsSetupExt Function

This function sets up a PDP context identified by its CID with some specific parameters.

- Prototype

```
s8 adl_gprsSetupExt ( u8                         Cid,
                      adl_gprsSetupParams_t      Params,
                      adl_port_e                 Port );
```

- Parameters

Cid:

The Cid of the PDP context to setup (integer value between 1 and 4).

Params:

The parameters to set up are contained in the following type:

```
typedef struct
{
   ascii* APN;
   ascii* Login;
   ascii* Password;
   ascii* FixedIP;
   bool   HeaderCompression;
   bool   DataCompression;
}adl_gprsSetupParams_t;
```

- APN:
  Address of the Provider GPRS Gateway (GGSN)
  maximum 100 bytes string

- Login:
  GPRS account login
  maximum 50 bytes string

- Password:
  GPRS account password
  maximum 50 bytes string

- o FixedIP:
  Optional fixed IP address of the MS (used only if not set to NULL) maximum 15 bytes string
- o HeaderCompression:
  PDP header compression option (enabled if set to TRUE)
- o DataCompression:
  PDP data compression option (enabled if set to TRUE)

Port:

Port on which to run the PDP context setup command. Setup return events are received in the GPRS event handler. If the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- Returned values

This function returns **OK** on success, or a negative error value.

Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	parameter error: bad Cid value or unavailable port
ADL_RET_ERR_PIN_KO	If the PIN is not entered, or if the "+WIND:4" indication has not occurred yet
ADL_GPRS_CID_NOT_DEFINED	problem to set up the Cid (the CID is already activated)
ADL_NO_GPRS_SERVICE	if the GPRS service is not supported by the product
ADL_RET_ERR_BAD_STATE	The service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function
ADL_RET_ERR_SERVICE_LOCKED	If the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.18.5 The adl_gprsAct Function

This function just runs the `adl_gprsActExt` one on the **ADL_PORT_OPEN_AT_VIRTUAL_BASE** port (cf. `adl_gprsActExt` description for more information). Please note that events generated by the `adl_gprsAct` will not be able to be forwarded to any external port, since the setup command was running on the Open AT® port.

3.18.6 The adl_gprsActExt Function

This function activates a specific PDP context identified by its Cid.

- Prototype

```
s8 adl_gprsActExt (u8             Cid,
                   adl_port_e     Port );
```

- Parameters

Cid:

The Cid of the PDP context to activate (integer value between 1 and 4).

Port:

Port on which to run the PDP context activation command. Activation return events are received in the GPRS event handler.If the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- Returned values

This function returns **OK** on success, or a negative error value.

Possible error values are:

Error Value	Description
ADL_RET_ERR_PARAM	parameters error: bad Cid value or unavailable port
ADL_RET_ERR_PIN_KO	If the PIN is not entered, or if the "+WIND:4" indication has not occurred yet
ADL_GPRS_CID_NOT_DEFINED	problem to set up the Cid (the CID is already activated)
ADL_NO_GPRS_SERVICE	if the GPRS service is not supported by the product
ADL_RET_ERR_BAD_STATE	The service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function
ADL_RET_ERR_SERVICE_LOCKED	If the function was called from a low level Interrupt handler (the function is forbidden in this context).

<u>Important Note:</u>

This function must be called before opening the GPRS FCM Flows.

3.18.7 The adl_gprsDeact Function

This function runs the `adl_gprsDeactExt` on the **ADL_PORT_OPEN_AT_VIRTUAL_BASE** port (cf. `adl_gprsDeactExt` description for more information). Please note that events generated by the `adl_gprsDeact` will not be able to be forwarded to any external port, since the setup command runs on the Open AT® port.

3.18.8 The adl_gprsDeactExt Function

This function deactivates a specific PDP context identified by its Cid.

- Prototype

```
s8 adl_gprsDeactExt ( u8            Cid
                      adl_port_e    Port );
```

- Parameters

Cid:

The Cid of the PDP context to deactivate (integer value between 1 and 4).

Port:

Port on which to run the PDP context deactivation command. Deactivation return events are received in the GPRS event handler.If the application requires ADL to forward these events, they will be forwarded to this Port parameter value.

- Returned values

This function returns **OK** on success, or a negative error value.

Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	parameters error: bad Cid value or unavailable port
ADL_RET_ERR_PIN_KO	if the PIN is not entered, or if the "+WIND:4" indication has not occurred yet
ADL_GPRS_CID_NOT_DEFINED	problem to set up the Cid (the CID is already activated)
ADL_NO_GPRS_SERVICE	if the GPRS service is not supported by the product
ADL_RET_ERR_BAD_STATE	the service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function
ADL_RET_ERR_SERVICE_LOCKED	If the function was called from a low level Interrupt handler (the function is forbidden in this context).

Important note:

If the GPRS flow is running, please do wait for the **ADL_FCM_EVENT_FLOW_CLOSED** event before calling the **adl_gprsDeact** function, in order to prevent Wireless CPU® lock.

3.18.9 The adl_gprsGetCidInformations Function

This function gets information about a specific activated PDP context identified by its Cid.

- Prototype

```
s8 adl_gprsGetCidInformations ( u8                    Cid,
                                adl_gprsInfosCid_t *  Infos );
```

- Parameters

Cid:

The Cid of the PDP context (integer value between 1 and 4).

Infos:

Information of the activated PDP context is contained in the following type:

```
typedef struct
{
    u32 LocalIP; // Local IP address of the MS
    u32 DNS1;    // First DNS IP address
    u32 DNS2;    // Second DNS IP address
    u32 Gateway; // Gateway IP address
}adl_gprsInfosCid_t;
```

This parameter fields will be set only if the GPRS session is activated; otherwise, they all will be set to 0.

- Returned values

This function returns **OK** on success, or a negative error value.

Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	parameters error: bad Cid value
ADL_RET_ERR_PIN_KO	if the PIN is not entered, or if the "+WIND:4" indication has not occurred yet
ADL_GPRS_CID_NOT_DEFINED	problem to set up the Cid (the CID is already activated)
ADL_NO_GPRS_SERVICE	if the GPRS service is not supported by the product
ADL_RET_ERR_BAD_STATE	the service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function

3.18.10 The adl_gprsUnsubscribe Function

This function unsubscribes from the GPRS service. The provided handler will not receive any more GPRS events.

- Prototype

```
s8 adl_gprsUnsubscribe ( adl_gprsHdlr_f Handler );
```

- Parameters

  Handler:

  Handler used with **adl_gprsSubscribe** function.

- Returned values

  This function returns **OK** on success, or a negative error value.

  Possible error values are:

Error value	Description
ADL_RET_ERR_PARAM	parameter error
ADL_RET_ERR_UNKNOWN_HDL	the provided handler is unknown
ADL_RET_ERR_NOT_SUBSCRIBED	the service is not subscribed
ADL_RET_ERR_BAD_STATE	the service is still processing another GPRS API ; application should wait for the corresponding event (indication of end of processing) in the GPRS handler before calling this function
ADL_RET_ERR_SERVICE_LOCKED	If the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.18.11 The adl_gprsIsAnIPAddress Function

This function checks if the provided string is a valid IP address. Valid IP address strings arebased on the "a.b.c.d" format, where a, b, c & d are integer values between 0 and 255.

- Prototype

```
bool adl_gprsIsAnIPAddress ( ascii * AddressStr );
```

- Parameters

  AddressStr:

  IP address string to check.

- Returned values
  - TRUE if the provided string is a valid IP address one, and FALSE otherwise.
  - **NULL** & empty string ("") are not considered as a valid IP address.

3.18.12 Example

This example just demonstrates how to use the GPRS service in a nominal case (error cases are not handled).

Complete examples using the GPRS service are also available on the SDK (Ping_GPRS sample).

```
// Global variables
adl_gprsSetupParams_t MyGprsSetup;
adl_gprsInfosCid_t    InfosCid;

// GPRS event handler
s8 MyGprsEventHandler ( u16 Event, u8 CID )
{
    // Trace event
    TRACE (( 1, "Received GPRS event %d/%d", Event, CID ));

    // Switch on event
    switch ( Event )
    {
        case ADL_GPRS_EVENT_SETUP_OK :
            TRACE (( 1, "PDP Ctxt Cid %d Setup OK", CID ));
            // Activate the session
            adl_gprsAct ( 1 );
        break;

        case ADL_GPRS_EVENT_ACTIVATE_OK :
            TRACE (( 1, "PDP Ctxt %d Activation OK", CID ));
            // Get context information
            adl_gprsGetCidInformations ( 1, &InfosCid );
            // De-activate the session
            adl_gprsDeAct ( 1 );
        }
        break;

        case ADL_GPRS_EVENT_DEACTIVATE_OK :
            TRACE (( 1, " PDP Ctxt %d De-activation OK", CID ));
            // Un-subscribe from GPRS event handler
           adl_gprsUnsubscribe ( MyGprsEventHandler );
        break;
    }

    // Forward event
    return ADL_GPRS_FORWARD;
}
```

```
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Fill Setup structure
    MyGprsSetup.APN = "myapn";
    MyGprsSetup.Login = "login";
    MyGprsSetup.Password = "password";
    MyGprsSetup.FixedIP = NULL;
    MyGprsSetup.HeaderCompression = FALSE;
    MyGprsSetup.DataCompression = FALSE;

    // Subscribe to GPRS event handler
    adl_gprsSubscribe ( MyGprsEventHandler );

    // Set up the GPRS context
    adl_gprsSetup ( 1, MyGprsSetup );
}
```

3.19 Semaphore ADL Service

The ADL Semaphore service allows the application to handle the semaphore resources supplied by the Open AT® OS.

Semaphores are used to synchronize processes between the application task and high level Interrupt handlers.

Note:

Semaphores cannot be used in a low level Interrupt handler context.

The defined operations are:

- A subscription function **adl_semSubscribe** to get a semaphore resource control
- An unsubscription function **adl_semUnsubscribe** to release a semaphore resource
- Consumption functions **adl_semConsume** and **adl_semConsumeDelay** to consume a semaphore counter
- A produce function **adl_semProduce** to produce a semaphore counter
- A test function **adl_semIsConsumed** to check a semaphore current state
- A capabilities function **adl_semGetResourcesCount** to retrieve the currently free semaphore resources count

3.19.1 Required Header File

The header file for the Semaphore service definitions is:

```
adl_ sem.h
```

3.19.2 The adl_semGetResourcesCount Function

This function retrieves the count of currently free semaphore resources for the application usage.

- Prototype

```
u32 adl_semGetResourcesCount ( void );
```

- Returned values
  - o Free semaphore resources count.

3.19.3 The adl_semSubscribe Function

This function allows the application to reserve and initialize a semaphore resource.

- Prototype

```
s32 adl_semSubscribe ( u16 SemCounter );
```

- Parameters

  SemCounter:

  Semaphore inner counter initialization value (reflects the number of times the semaphore can be consumed before the calling task must be suspended).

- Returned values
  - o **Handle** A positive semaphore service handle on success:
    - Semaphore service handle, to be used in further service function calls.
  - o A negative error value otherwise:
    - **ADL_RET_ERR_NO_MORE_SEMAPHORES** when there are no more free semaphore resources.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.19.4 The adl_semConsume Function

This function allows the application to reduce the required semaphore counter by one.
If this counter value falls under zero, the calling execution context is suspended until the semaphore is produced from another context.

- Prototype

  `s32 adl_semConsume ( s32 SemHandle );`

- Parameters

  SemHandle:

  Semaphore service handle, previously returned by the **adl_semSubscribe** function.

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_UNKNOWN_HDL** when the supplied handle is unknown.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

- Exceptions

  The following exception must be generated on this function call
  - o 205 If the semaphore has been consumed too many times. A semaphore can be consumed a number of times equal to its initial value + 256.

3.19.5 The adl_semConsumeDelay Function

This function allows the application to reduce the required semaphore counter by one.

If this counter value falls under zero, the calling execution context is suspended until the semaphore is produced from another context. Moreover, if the semaphore is not produced during the supplied time-out duration, the calling context is automatically resumed.

- Prototype

```
s32 adl_semConsumeDelay (s32        SemHandle,
                         u32        TimeOut );
```

- Parameters

  SemHandle:

  Semaphore service handle, previously returned by the **adl_semSubscribe** function.

  Timeout:

  Time to wait before resuming context when the semaphore is not produced (must not be 0). Time measured is in 18.5 ms ticks.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** when the supplied handle is unknown.
  - **ADL_RET_ERR_PARAM** when a supplied parameter value is wrong.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

- Exceptions

  The following exception must be generated on this function call.

  - 206 if the semaphore has been consumed too many times.
    A semaphore can be consumed a number of times equal to its initial value + 256.

3.19.6 The adl_semProduce Function

This function allows the application to increase the required semaphore counter by one.
If this counter value gets above zero, the execution contexts that were suspended due to using this semaphore are resumed.

- Prototype

```
s32 adl_semProduce ( s32   SemHandle );
```

- Parameters

  SemHandle:

  Semaphore service handle, previously returned by the **adl_semSubscribe** function.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied handle is unknown.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

- Exceptions

  The following exception must be generated on this function call.
  - 133 if the semaphore has been produced too many times.
    A semaphore can be produced until its inner counter reaches its initial value.

3.19.7 The adl_semUnsubscribe Function

This function allows the application to unsubscribe from the Semaphore service, in order to release the previously reserved resource.
A semaphore can be unsubscribed only if its inner counter value is the initial one (the semaphore has been produced as many times as it has been consumed).

- Prototype

```
s32 adl_semUnsubscribe ( s32   SemHandle );
```

- Parameters

  SemHandle:

  Semaphore service handle, previously returned by the **adl_semSubscribe** function.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** when the supplied handle is unknown
  - **ADL_RET_ERR_BAD_STATE** when the semaphore inner counter value is different from the initial value.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.19.8 The adl_semIsConsumed Function

This function allows the application to check if a semaphore is currently consumed (the internal counter value is lower than the initial value) or not (the counter value is the initial one).

- Prototype

```
s32 adl_semIsConsumed ( s32   SemHandle );
```

- Parameters

  SemHandle:

  Semaphore service handle, previously returned by the **adl_semSubscribe** function.

- Returned values
  - TRUE if the semaphore resource is consumed.
  - FALSE If the semaphore resource is not consumed.
  - **ADL_RET_ERR_UNKNOWN_HDL** when the supplied handle is unknown.

3.19.9 Example

This example shows how to use the Semaphore service in a nominal case (error cases are not handled).

```
// Global variable: Semaphore service handle
s32 MySemHandle;

// Somewhere in the application code, used as high level interrupt handler
void MyHighLevelHandler ( void )
{
    // Produces the semaphore, to resume the application task context
    adl_semProduce ( MySemHandle );
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribes to the semaphore service
    MySemHandle = adl_semSubscribe ( 0 );

    // Consumes the semaphore, with a 37 ms time-out delay
    adl_semConsumeDelay ( MySemHandle, 2 );

    // Consumes the semaphore: has to be produced from another context
    adl_semConsume ( MySemHandle );

void MyFunction2 ( void )
{
    // Un-subscribes from the semaphore service
    adl_semUnsubscribe ( MySemHandle );
}
```

3.20 Application Safe Mode Service

By default, the +WOPEN and +WDWL commands cannot be filtered by any embedded application. This service allows one application to get these commands events, in order to prevent any external application stop or erase the current embedded one.

3.20.1 Required Header File

The header file for the Application safe mode service is:

```
adl_safe.h
```

3.20.2 The adl_safeSubscribe Function

This function subscribes to the Application safe mode service in order to receive +WOPEN and +WDWL commands events.

- Prototype

```
s8    adl_safeSubscribe (    u16              WDWLopt,
                             u16              WOPENopt,
                             adl_safeHdlr_f   SafeHandler );
```

- Parameters

WDWLopt:

Additionnal options for +WDWL command subscription. This command is at least subscribed in ACTION and READ mode. Please see 3.3.4.6 **adl_atCmdSubscribe** API for more details about these options.

WOPENopt:

Additionnal options for +WOPEN command subscription. This command is at least subscribed in READ, TEST and PARAM mode, with minimum of one mandatory parameter. Please see 3.3.4.6 **adl_atCmdSubscribe** API for more details about these options.

SafeHandler:

Application safe mode handler defined using the following type:

```
typedef bool (*adl_safeHdlr_f) (adl_safeCmdType_e      CmdType,
                                adl_atCmdPreParser_t *  paras );
```

The CmdType events received by this handler are defined below:

```
typedef enum
{
  ADL_SAFE_CMD_WDWL,                // AT+WDWL command
  ADL_SAFE_CMD_WDWL_READ,           // AT+WDWL? command
  ADL_SAFE_CMD_WDWL_OTHER,          // WDWL other syntax
  ADL_SAFE_CMD_WOPEN_STOP,          // AT+WOPEN=0 command
  ADL_SAFE_CMD_WOPEN_START,         // AT+WOPEN=1 command
  ADL_SAFE_CMD_WOPEN_GET_VERSION,   // AT+WOPEN=2 command
  ADL_SAFE_CMD_WOPEN_ERASE_OBJ,     // AT+WOPEN=3 command
  ADL_SAFE_CMD_WOPEN_ERASE_APP,     // AT+WOPEN=4 command
  ADL_SAFE_CMD_WOPEN_SUSPEND_APP,   // AT+WOPEN=5 command
  ADL_SAFE_CMD_WOPEN_AD_GET_SIZE,   // AT+WOPEN=6 command
  ADL_SAFE_CMD_WOPEN_AD_SET_SIZE,   // AT+WOPEN=6,<size> command
  ADL_SAFE_CMD_WOPEN_READ,          // AT+WOPEN? command
  ADL_SAFE_CMD_WOPEN_TEST,          // AT+WOPEN=? command
  ADL_SAFE_CMD_WOPEN_OTHER          // WOPEN other syntax
} adl_safeCmdType_e;
```

The **paras** received structure contains the same parameters as the commands used for **adl_atCmdSubscribe** API.

If the Handler returns FALSE, the command will not be forwarded to the Wavecom Firmware.

If the Handler returns TRUE, the command will be processed by the Wavecom Firmware, which will send responses to the external application.

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_PARAM** if the parameters have an incorrect value
  - o **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service is already subscribed
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.20.3 The adl_safeUnsubscribe Function

This function unsubscribes from Application safe mode service. The +WDWL and +WOPEN commands are not filtered anymore and are processed by the Wavecom Firmware.

- Prototype

```
s8    adl_safeUnsubscribe    ( adl_safeHdlr_f  Handler);
```

- Parameters

Handler:

Handler used with **adl_safeSubscribe** function.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** if the parameter has an incorrect value
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handler is unknown
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if the service is not subscribed
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level Interrupt handler (the function is forbidden in this context).

3.20.4 The adl_safeRunCommand Function

This function allows +WDWL or +WOPEN command with any standard syntax.

- Prototype

```
s8    adl_safeRunCommand (adl_safeCmdType_e     CmdType,
                          adl_atRspHandler_t    RspHandler );
```

- Parameters

CmdType:

Command type to run; please refer to **adl_safeSubscribe** description. **ADL_SAFE_CMD_WDWL_OTHER** and **ADL_SAFE_CMD_WOPEN_OTHER** values are not allowed.

The **ADL_SAFE_CMD_WOPEN_SUSPEND_APP** may be used to suspend the Open AT® application task. The execution may be resumed using the AT+WOPENRES command, or by sending a signal on the hardware Interrupt product pin (The INTERRUPT feature has to be enabled on the product: please refer to the AT+WFM command). Open AT® application running in Remote Task Environment cannot be suspended (the function has no effect). Please note that the current Open AT® application process is suspended immediately on the **adl_safeRunCommand** process; if there is any code after this function call, it will be executed only when the process is resumed.

RspHandler:

Response handler to get command results. All responses are subscribed and the command is executed on the Open AT® virtual port. Instead of providing a response handler, a port identifier may be specified (using adl_port_e type): the command will be executed on this port, and the resulting responses sent back on this port.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** if the parameter has an incorrect value
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.21 AT Strings Service

This service provides APIs to process AT standard response strings.

3.21.1 Required Header File

The header file for the AT strings service is:

```
adl_str.h
```

3.21.2 The adl_strID_e Type

All predefined AT strings for this service are defined in the following type:

```
typedef enum
{
  ADL_STR_NO_STRING,                    // Unknown string
  ADL_STR_OK,                           // "OK"
  ADL_STR_BUSY,                         // "BUSY"
  ADL_STR_NO_ANSWER,                    // "NO ANSWER"
  ADL_STR_NO_CARRIER,                   // "NO CARRIER"
  ADL_STR_CONNECT,                      // "CONNECT"
  ADL_STR_ERROR,                        // "ERROR"
  ADL_STR_CME_ERROR,                    // "+CME ERROR:"
  ADL_STR_CMS_ERROR,                    // "+CMS ERROR:"
  ADL_STR_CPIN,                         // "+CPIN:"
  ADL_STR_LAST_TERMINAL,                // Terminal resp. are
                                           before this line
  ADL_STR_RING = ADL_STR_LAST_TERMINAL, // "RING"
  ADL_STR_WIND,                         // "+WIND:"
  ADL_STR_CRING,                        // "+CRING:"
  ADL_STR_CPINC,                        // "+CPINC:"
  ADL_STR_WSTR,                         // "+WSTR:"
  ADL_STR_CMEE,                         // "+CMEE:"
  ADL_STR_CREG,                         // "+CREG:"
  ADL_STR_CGREG,                        // "+CGREG:"
  ADL_STR_CRC,                          // "+CRC:"
  ADL_STR_CGEREP,                       // "+CGEREP:"
  ADL_STR_LAST                          // Last string ID
} adl_strID_e;
```

3.21.3 The adl_strGetID Function

This function returns the ID of the provided response string.

- Prototype

```
adl_strID_e adl_strGetID  (ascii *    rsp );
```

- Parameters

rsp:

String to parse to get the ID.

- Returned values

  - **ADL_STR_NO_STRING** if the string is unknown.
  - **Id** of the string otherwise.

3.21.4 The adl_strGetIDExt Function

This function returns the ID of the provided response string, with an optional argument and its type.

- Prototype

```
adl_strID_e adl_strGetIDExt   (ascii *    rsp
                               void *     arg
                               u8 *       argtype );
```

- Parameters

rsp:

String to parse to get the ID.

arg:

Parsed first argument; not used if set to NULL.

argtype:

Type of the parsed argument:

if argtype is **ADL_STR_ARG_TYPE_ASCII**, arg is an **ascii *** string ;

if argtype is **ADL_STR_ARG_TYPE_U32**, arg is an **u32 *** integer.

- Returned values

  - **ADL_STR_NO_STRING** if the string is unknown.
  - **Id** of the string otherwise.

3.21.5 The adl_strIsTerminalResponse Function

This function checks whether the provided response ID is a terminal one. A terminal response is the last response that a response handler will receive from a command.

- Prototype

```
bool  adl_strIsTerminalResponse (adl_strID_e   RspID );
```

- Parameters

RspID:

Response ID to check.

- Returned values
  - `TRUE` if the provided response ID is a terminal one.
  - `FALSE` otherwise.

3.21.6 The adl_strGetResponse Function

This function provides the standard response string from its ID.

- Prototype

```
ascii *    adl_strGetResponse (adl_strID_e     RspID );
```

- Parameters

RspID:

Response ID from which to get the string.

- Returned values
  - Standard response string on success ;
  - `NULL` if the ID does not exist.

Important caution:

The returned pointer memory is allocated by this function, but its ownership is transferred to the embedded application. This means that the embedded application will have to release the returned pointer.

3.21.7 The adl_strGetResponseExt Function

This function provides a standard response string from its ID, with the provided argument.

- Prototype

```
ascii *    adl_strGetResponseExt   (adl_strID_e      RspID,
                                     u32             arg );
```

- Parameters

RspID:

Response ID from which to get the string.

arg:

Response argument to copy in the response string. Depending on the response ID, this argument should be an **u32** integer value, or an **ascii *** string.

- Returned values
  - o Standard response string on success ;
  - o **NULL** if the ID does not exist.

Important caution:

The returned pointer memory is allocated by this function, but its ownership is transferred to the embedded application. This means that the embedded application will have to release the returned pointer.

3.22 Application & Data Storage Service

This service provides APIs to use the Application & Data storage volume. This volume may be used to store data, or ".dwl" files (Wavecom Firmware updates, new Open AT® applications or E2P configuration files) in order to be installed later on the product.

The default storage size is 768 Kbytes. It may be configured with the AT+WOPEN command (Please refer to the AT commands interface guide (document [1]) for more information).

This storage size has to be set to the maximum (about 1.2 Mbytes) in order to have enough place to store a Wavecom Firmware update.

Caution:

Any A&D size change will lead to an area format process (some additional seconds on start-up, all A&D cells data will be erased).

Legal mention:

The Download Over The Air feature enables the Wavecom Firmware to be remotely updated.

The downloading and OS updating processes have to be activated and managed by an appropriate Open AT® based application to be developed by the customer. The security of the whole process (request for update, authentication, encryption, etc) has to be managed by the customer under his own responsibility. Wavecom shall not be liable for any issue related to any use by customer of the Download Over The Air feature.

Wavecom AGREES AND THE CUSTOMER ACKNOWLEDGES THAT THE SDK Open AT® IS PROVIDED "AS IS" BY Wavecom WITHOUT ANY WARRANTY OR GUARANTEE OF ANY KIND.

3.22.1 Required Header File

The header file for the Application & Data storage service is:

```
adl_ad.h
```

3.22.2 The adl_adSubscribe Function

This function subscribes to the required A&D space cell identifier.

- Prototype

```
s32    adl_adSubscribe   (u32        CellID
                           u32        Size );
```

- Parameters

  CellID:

  A&D space cell identifier to subscribe to. This cell may already exist or not. If the cell does not exist, the given size is allocated.

  Size:

  New cell size in bytes (this parameter is ignored if the cell already exists). It may be set to **ADL_AD_SIZE_UNDEF** for a variable size. In this case, new cells subscription will fail until the undefined size cell is finalised.

  Total used size in flash will be the data size + header size. Header size is variable (with an average value of 16 bytes).

  When subscribing, the size is rounded up to the next multiple of 4.

- Returned values
  - A positive or null value on success:
    - The A&D cell handle on success, to be used on further A&D API functions calls,
  - A negative error value:
    - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the cell is already subscribed;
    - **ADL_AD_RET_ERR_OVERFLOW** if there is not enough allocated space,
    - **ADL_AD_RET_ERR_NOT_AVAILABLE** if there is no A&D space available on the product,
    - **ADL_RET_ERR_PARAM** if the CellId parameter is 0xFFFFFFFF (this value should not be used as an A&D Cell ID),
    - **ADL_RET_ERR_BAD_STATE** (when subscribing an undefined size cell) if another undefined size cell is already subscribed and not finalized.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.3 The adl_adUnsubscribe Function

This function unsubscribes from the given A&D cell handle.

- Prototype

```
s32   adl_adUnsubscribe  (s32   CellHandle );
```

- Parameters

  CellHandle:

  A&D cell handle returned by **adl_adSubscribe** function.

- Returned values
  - **OK** on success,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle was not subscribed.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.4 The adl_adEventSubscribe Function

This function allows the application to provide ADL with an event handler to be notified with A&D service related events.

- Prototype

```
s32   adl_adEventSubscribe   (adl_adEventHdlr_f      Handler );
```

- Parameters

  Handler:

  Call-back function provided by the application. Please refer to next chapter for more information.

- Returned values
  - o A positive or null value on success:
    - A&D event handle, to be used in further A&D API functions calls,
  - o A negative error value:
    - **ADL_RET_ERR_PARAM** if the **Handler** parameter is invalid,
    - **ADL_RET_ERR_NO_MORE_HANDLES** if the A&D event service has been subscribed more than 128 times.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note:

In order to format or re-compact the A&D storage volume, the **adl_adEventSubscribe** function has to be called before the **adl_adFormat** or the **adl_adRecompact** functions.

3.22.5 The adl_adEventHdlr_f Call-back Type

This call-back function has to be provided to ADL through the **adl_adEventSubscribe** interface, in order to receive A&D related events.

- Prototype

```
typedef void (*adl_adEventHdlr_f) ( adl_adEvent_e     Event,
                                    u32               Progress );
```

- Parameters

Event:

Event is the received event identifier. The events (defined in the **adl_adEvent_e** type) are described in the table below.

Event	Meaning
ADL_AD_EVENT_FORMAT_INIT	The **adl_adFormat** function has been called by an application (a format process has just been required).
ADL_AD_EVENT_FORMAT_PROGRESS	The format process is on going. Several "progress" events should be received until the process is completed.
ADL_AD_EVENT_FORMAT_DONE	The format process is over. The A&D storage area is now usable again. All cells have been erased, and the whole storage place is available.
ADL_AD_EVENT_RECOMPACT_INIT	The **adl_adRecompact** function has been called by an application (a re-compaction process has been required).
ADL_AD_EVENT_RECOMPACT_PROGRESS	The re-compaction process is on going. Several "progress" events should be received until the process is completed.
ADL_AD_EVENT_RECOMPACT_DONE	The re-compaction process is over: the A&D storage area is now usable again. The space previously used by deleted cells is now free.
ADL_AD_EVENT_INSTALL	The **adl_adInstall** function has been called by an application (an install process has just been required and the Wireless CPU® is going to reset).

Progress:

On **ADL_AD_EVENT_FORMAT_PROGRESS** & **ADL_AD_EVENT_RECOMPACT_PROGRESS** events reception, this parameter is the process progress ratio (considered as a percentage).

On **ADL_AD_EVENT_FORMAT_DONE** & **ADL_AD_EVENT_RECOMPACT_DONE** events reception, this parameter is set to 100%.

Otherwise, this parameter is set to 0.

3.22.6 The adl_adEventUnsubscribe Function

This function allows the Open AT® application to unsubscribe from the A&D events notification.

- Prototype

```
s32 adl_adEventUnsubscribe (s32    EventHandle );
```

- Parameters

  EventHandle:

  Handle previously returned by the **adl_adEventSubscribe** function.

- Returned values

  - **OK** on success,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown,
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if no A&D event handler has been subscribed,
  - **ADL_RET_ERR_BAD_STATE** if a format or re-compaction process is currently running with this event handle.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.7 The adl_adWrite Function

This function writes data at the end of the given A&D cell.

- Prototype

```
s32   adl_adWrite( s32       CellHandle
                   u32       Size
                   void *    Data   );
```

- Parameters

  CellHandle:

  A&D cell handle returned by **adl_adSubscribe** function.

  Size:

  Data buffer size in bytes.

  Data:

  Data buffer.

- Returned values

  - **OK** on success ;
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle was not subscribed ;
  - **ADL_RET_ERR_PARAM** on parameter error ;
  - **ADL_RET_ERR_BAD_STATE** if the cell is finalized ;
  - **ADL_AD_RET_ERR_OVERFLOW** if the write operation exceeds the cell size.

- o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.8 The adl_adInfo Function

This function provides information on the requested A&D cell.

- Prototype

```
s32   adl_adInfo  ( s32             CellHandle
                    adl_adInfo_t *  Info );
```

- Parameters

CellHandle:

A&D cell handle returned by `adl_adSubscribe` function.

Info:

Information structure on requested cell, based on following type:

```
typedef struct
{
   u32   identifier;   // identifier
   u32   size;         // entry size
   void  *data;        // pointer to stored data
   u32   remaining;    // remaining writable space unless finalized
   bool  finalised;    // TRUE if entry is finalized
}adl_adInfo_t;
```

- Returned values
  - o **OK** on success,
  - o **ADL_RET_ERR_PARAM** on parameter error,
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the handle was not subscribed,
  - o **ADL_RET_ERR_BAD_STATE** if the required cell is a not finalized or an undefined size.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.9 The adl_adFinalise Function

This function set the provided A&D cell in read-only (finalized) mode. The cell content can not be modified.

- Prototype

```
s32   adl_adFinalise   (s32  CellHandle );
```

- Parameters

CellHandle:

A&D cell handle returned by `adl_adSubscribe` function.

- Returned values
  - **OK** on success,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle was not subscribed,
  - **ADL_RET_ERR_BAD_STATE** if the cell was already finalized.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.10 The adl_adDelete Function

This function deletes the provided A&D cell. The used space and the ID will be available on next re-compaction process.

- Prototype

```
s32   adl_adDelete     (s32  CellHandle );
```

- Parameters

  CellHandle:

  A&D cell handle returned by `adl_adSubscribe` function.

- Returned values
  - **OK** on success,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle was not subscribed.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note:

Calling **adl_adDelete** will unsubscribe the allocated handle.

3.22.11 The adl_adInstall Function

This function installs the content of the requested cell, if it is a **.DWL** file. This file should be an Open AT® application, an EEPROM configuration file, an XModem downloader binary file, or a Wavecom Firmware binary file.

Caution:

This API resets the Wireless CPU® on success.

- Prototype

```
s32   adl_adInstall    (s32  CellHandle );
```

- Parameters

  CellHandle:

  A&D cell handle returned by **adl_adSubscribe** function.

- Returned values
  - Wireless CPU® resets on success. The parameter of the **adl_main** function is then set to **ADL_INIT_DOWNLOAD_SUCCESS**, or **ADL_INIT_DOWNLOAD_ERROR**, according to the **.DWL** file update success or

not.
Before the Wireless CPU® reset, all subscribed event handlers (if any) will receive the **ADL_AD_EVENT_INSTALL** event, in order to let them perform last operations.

- **ADL_RET_ERR_BAD_STATE** if the cell is not finalized,
- **ADL_RET_ERR_UNKNOWN_HDL** if the handle was not subscribed.
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

<u>Note for RTE:</u>

In RTE mode, calling this API will cause a message box display, prompting the user for installing the desired A&D cell content or not (see Figure 8: A&D cell content install window).

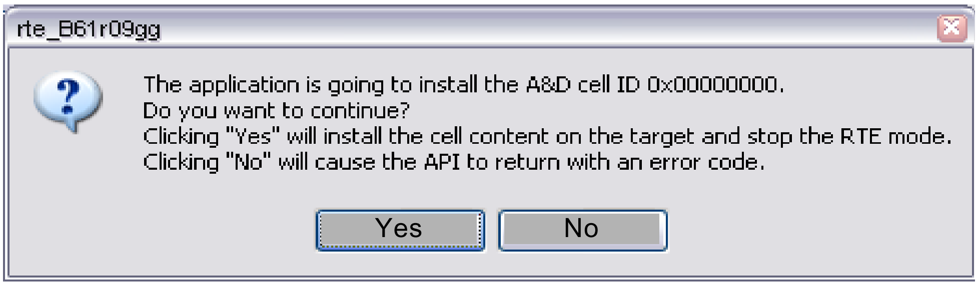


Figure 8: A&D cell content install window

If the user selects "No", the API will fail and return the ADL_AD_RET_ERROR code. If the user selects "Yes", the cell content is installed, the Wireless CPU® resets, and the RTE mode is automatically closed.

3.22.12 The adl_adRecompact Function

This function starts the re-compaction process, which will release the deleted cell spaces and IDs.

<u>Caution:</u>

If some A&D cells are deleted, and the recompaction process is not performed regularly, the deleted cell space will not be freed.

- Prototype

```
s32   adl_adRecompact   (s32  EventHandle );
```

- Parameters

  EventHandle:

  Event handle previously returned by the **adl_adEventSubscribe** function. The associated handler will receive the re-compaction process events sequence.

- Returned values
  - **OK** on success. Event handlers will receive the following event sequence:

  - **ADL_AD_EVENT_RECOMPACT_INIT** just after the process is launched,
  - **ADL_AD_EVENT_RECOMPACT_PROGRESS** several times, indicating the process progression,
  - **ADL_AD_EVENT_RECOMPACT_DONE** when the process is completed.
- **ADL_RET_ERR_BAD_STATE** if a re-compaction or format process is currently running,
- **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown,
- **ADL_RET_ERR_NOT_SUBSCRIBED** if no A&D event handler has been subscribed,
- **ADL_AD_RET_ERR_NOT_AVAILABLE** if there is no A&D space available on the product.
- **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.13 The adl_adGetState Function

This function provides an information structure on the current A&D volume state.

- Prototype

```
s32   adl_adGetState   ( adl_adState_t * State );
```

- Parameters

State:

A&D volume information structure, based on the following type:

```
typedef struct
{
    u32 freemem;          // Space free memory size
    u32 deletedmem;       // Deleted memory size
    u32 totalmem;         // Total memory
    u16 numobjects;       // Number of allocated objects
    u16 numdeleted;       // Number of deleted objects
    u8  pad;              // not used
} adl_adState_t;
```

- Returned values
  - **OK** on success,
  - **ADL_AD_RET_ERR_NOT_AVAILABLE** if there is no A&D space available on the product
  - **ADL_AD_RET_ERR_NEED_RECOMPACT** if a power down or a reset occurred when a re-compaction process was running. The application has to launch the **adl_adRecompact** function before using any other A&D service function.
  - **ADL_RET_ERR_PARAM** on parameter error.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.14 The adl_adGetCellList Function

This function provides the list of the current allocated cells.

- Prototype

```
s32   adl_adGetCellList ( wm_lst_t * CellList );
```

- Parameters

CellList:

Return allocated cell list. The list elements are the cell identifiers and are based on u32 type.

The list is ordered by cell id values, from the lowest to the highest.

Caution:

The list memory is allocated by the adl_adGetCellList function and has to be released with the wm_lstDestroy function by the application.

- Returned values
    - **OK** on success ;
    - **ADL_AD_RET_ERR_NOT_AVAILABLE** if there is no A&D space available on the product ;
    - **ADL_RET_ERR_PARAM** on parameter error.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note:

- The number of elements in the returned list are limited by **ADL_AD_MAX_CELL_RETRIEVE;**
- If the number of cell IDs to get is superior to **ADL_AD_MAX_CELL_RETRIEVE**, use **adl_adFindInit** and **adl_adFindNext** functions.

3.22.15 The adl_adFormat Function

This function re-initializes the A&D storage volume. It is only allowed if there is currently no subscribed cells, or if there are no currently running re-compaction or format process.

Important caution:

All the A&D storage cells will be erased by this operation. The A&D storage format process can take several seconds.

- Prototype

```
s32   adl_adFormat (s32      EventHandle );
```

- Parameters

EventHandle:

Event handle previously returned by the **adl_adEventSubscribe** function. The associated handler will receive the format process events sequence

- Returned values
  - **OK** on success. Event handlers will receive the following event sequence:
    - **ADL_AD_EVENT_FORMAT_INIT** just after the process is launched,
    - **ADL_AD_EVENT_FORMAT_PROGRESS** several times, indicating the process progression,
    - **ADL_AD_EVENT_FORMAT_DONE** once the process is done,
  - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown,
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if no A&D event handler has been subscribed,
  - **ADL_AD_RET_ERR_NOT_AVAILABLE** if there is no A&D space available on the product,
  - **ADL_RET_ERR_BAD_STATE** if there is at least one currently subscribed cell, or if a re-compaction or format process is already running.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.22.16 The adl_adFindInit Function

This function initializes a cell search between the two provided cell identifiers.

- Prototype

```
s32   adl_adFindInit (u32                 MinCellId,
                      u32                 MaxCellId,
                      adl_adBrowse_t*     BrowseInfo );
```

- Parameters

MinCellId:

Minimum cell value for wanted cell identifiers.

MaxCellId:

Maximum cell value for wanted cell identifiers.

BrowseInfo:

Returned browse information, to be used with the adl_adFindNext function. Based on the following type:

```
typedef struct
{
  u32       hidden[4];   // Memory space necessary for cell information
}adl_adBrowse_t;
```

- Returned values
  - **OK** on success.
  - **ADL_AD_RET_ERR_NOT_AVAILABLE** if A&D space is not available
  - **ADL_RET_ERR_PARAM** on parameter error.

3.22.17 The adl_adFindNext Function

This function performs a cell ID search on the browse informations provided by the **adl_adFindInit** function.

- Prototype

```
s32   adl_adFindNext (adl_adBrowse_t*     BrowseInfo,
                      u32*                CellId );
```

- Parameters

  BrowseInfo:

  Browse informations, returned by the **adl_adFindInit** function.

  CellId:

  Next found Cell ID.

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_PARAM** on parameter error.
  - o **ADL_AD_RET_REACHED_END** no more elements to enumerate.

3.22.18 Example

This example demonstrates how to use the A&D service in a nominal case (error cases not handled).

Complete examples using the A&D service are also available on the SDK (DTL Application_Download sample, generic Download library sample).

```
// Global variables & constants

// Cell & event handles
s32 MyADCellHandle;
s32 MyADEventHandle;

// Info & state structure
adl_adInfo_t Info;
adl_adState_t State;

// A&D event handler
void MyADEventHandler ( adl_adEvent_e Event, u32 Progress )
{
    // Check event
    switch ( Event )
    {
        case ADL_AD_EVENT_RECOMPACT_DONE :
        case ADL_AD_EVENT_FORMAT_DONE :
            // The process is over
   TRACE (( 1, "Format/Recompact process over…" ));
        break;
    }
}
```

```
...

// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    u8 DataBuffer [ 10 ];

    // Get state
    adl_adGetState ( &State );

    // Subscribe to the A&D event service
    MyADEventHandle = adl_adEventSubscribe ( MyADEventHandler );

    // Subscribe to an A&D cell
    MyADCellHandle = adl_adSubscribe ( 0x00000000, 20 );

    // Write data buffer
    wm_memset ( DataBuffer, 0, 10 );
    adl_adWrite ( MyADCellHandle, 10, DataBuffer );

    // Get info
    adl_adInfo ( MyADCellHandle, &Info );

    // Install the cell (will fail, not finalized)
    adl_adInstall ( MyADCellHandle );

    // Finalize the cell
    adl_adFinalise ( MyADCellHandle );

    // Delete the cell
    adl_adDelete ( MyADCellHandle );

    // Launch the re-compaction process
    adl_adRecompact ( MyADEventHandle );

    // Launch the format process
   // (will fail, re-compaction process is running)
    adl_adFormat ( MyADEventHandle );

    // Unsubscribe from the A&D event service
    // (will fail, re-compaction process is running)
    adl_adEventUnsubscribe ( MyADEventHandler );
}
```

3.23 AT/FCM IO Ports Service

ADL applications may use this service to be informed about the product AT/FCM IO ports states.

3.23.1 Required Header File

The header file for the AT/FCM IO Ports service is:

```
adl_port.h
```

3.23.2 AT/FCM IO Ports

AT Commands and FCM services can be used to send and receive AT Commands or data blocks, to or from one of the product ports. These ports are linked either to product physical serial ports (as UART1 / UART2 / USB ports), or virtual ports (as Open AT® virtual AT port, GSM CSD call data port, GPRS session data port or Bluetooth virtual ports).

AT/FCM IO Ports are identified by the type below:

```
typedef enum
{
    ADL_PORT_NONE,
    ADL_PORT_UART1,
    ADL_PORT_UART2,
    ADL_PORT_USB,

    ADL_PORT_UART1_VIRTUAL_BASE       = 0x10,
    ADL_PORT_UART2_VIRTUAL_BASE       = 0x20,
    ADL_PORT_USB_VIRTUAL_BASE         = 0x30,
    ADL_PORT_BLUETOOTH_VIRTUAL_BASE   = 0x40,
    ADL_PORT_GSM_BASE                 = 0x50,
    ADL_PORT_GPRS_BASE                = 0x60,
    ADL_PORT_OPEN_AT_VIRTUAL_BASE     = 0x80
} adl_port_e;
```

The available ports are described hereafter:

- **ADL_PORT_NONE**
  Not usable
- **ADL_PORT_UART1**
  Product physical UART 1
  Please refer to the AT+WMFM command documentation to know how to open/close this product port.

- `ADL_PORT_UART2`
  Product physical UART 2
  Please refer to the AT+WMFM command documentation to know how to open/close this product port.
- `ADL_PORT_USB`
  Product physical USB port (reserved for future products)
- `ADL_PORT_UART1_VIRTUAL_BASE`
  Base ID for 27.010 protocol logical channels on UART 1
  Please refer to AT+CMUX command & 27.010 protocol documentation to know how to open/close such a logical channel.
- `ADL_PORT_UART2_VIRTUAL _BASE`
  Base ID for 27.010 protocol logical channels on UART 2
  Please refer to AT+CMUX command & 27.010 protocol documentation to know how to open/close such a logical channel.
- `ADL_PORT_USB_VIRTUAL _BASE`
  Base ID for 27.010 protocol logical channels on USB link (reserved for future products)
- `ADL_PORT_BLUETOOTH_VIRTUAL _BASE`
  Base ID for connected Bluetooth peripheral virtual port.
  ONLY USABLE WITH THE FCM SERVICE
  Please refer to the Bluetooth AT commands documentation to know how to connect, and how to open/close such a virtual port.
- `ADL_PORT_GSM_BASE`
  Virtual Port ID for GSM CSD data call flow
  ONLY USABLE WITH THE FCM SERVICE
  Please note that this port will be considered as always available (no OPEN/CLOSE events for this port ; `adl_portIsAvailable` *function will always return TRUE)*
- `ADL_PORT_GPRS_BASE`
  Virtual Port ID for GPRS data session flow
  ONLY USABLE WITH THE FCM SERVICE
  Please note that this port will be considered as always available (no OPEN/CLOSE events for this port ; `adl_portIsAvailable` *function will always return TRUE) if the GPRS feature is supported on the current product.*
- `ADL_PORT_OPEN_AT_VIRTUAL_BASE`
  Base ID for AT commands contexts dedicated to Open AT® applications
  ONLY USABLE WITH THE AT COMMANDS SERVICE
  This port is always available, and is opened immediately at the product's start-up. This is the default port where are executed the AT commands sent by the AT Command service.

3.23.3 Ports Test Macros

Some ports & events test macros are provided. These macros are defined hereafter.

- `ADL_PORT_IS_A_SIGNAL_CHANGE_EVENT(_e)`
  Returns TRUE if the event "_e" is a signal change one, FALSE otherwise.
- `ADL_PORT_GET_PHYSICAL_BASE(_port)`
  Extracts the physical port identifier part of the provided "_port".

E.g. if used on a 27.010 virtual port identifier based on the UART 2, this macro will return ADL_PORT_UART2.

- **`ADL_PORT_IS_A_PHYSICAL_PORT(_port)`**
  Returns TRUE if the provided "_port" is a physical output based one (E.g. UART1, UART2 or 27.010 logical ports), FALSE otherwise.
- **`ADL_PORT_IS_A_PHYSICAL_OR_BT_PORT(_port)`**
  Returns TRUE is the provided "_port" is a physical output or a bluetooth based one, FALSE otherwise.
- **`ADL_PORT_IS_AN_FCM_PORT(_port)`**
  Returns TRUE if the provided "_port" is able to handle the FCM service (i.e. all ports except the Open AT® virtual base ones), FALSE otherwise.
- **`ADL_PORT_IS_AN_AT_PORT(_port)`**
  Returns TRUE if the provided "_port" is able to handle AT commands services (i.e. all ports except the GSM & GPRS virtual base ones), FALSE otherwise.

3.23.4 The adl_portSubscribe Function

This function subscribes to the AT/FCM IO Ports service in order to receive specific ports related events.

- Prototype

```
s8    adl_portSubscribe (adl_portHdlr_f   PortHandler );
```

- Parameters

PortHandler:

Port related events handler defined using the following type:

```
typedef void (*adl_portHdlr_f) (adl_portEvent_e      Event,
                                adl_port_e           Port,
                                u8                   State );
```

The events received by this handler are defined below:

- **`ADL_PORT_EVENT_OPENED`**
  *Informs the ADL application that the specified **Port** is now opened. According to its type, it may now be used with either AT Commands service or FCM service.*
- **`ADL_PORT_EVENT_CLOSED`**
  *Informs the ADL application that the specified **Port** is now closed. It is not usable anymore with neither AT Commands service nor FCM service.*
- **`ADL_PORT_EVENT_DSR_STATE_CHANGE`**
  *Informs the ADL application that the specified **Port** DSR signal state has changed to the new **State** value (0/1). This event will be received by all subscribers which have started a polling process on the specified **Port** DSR signal with the adl_portStartSignalPolling function.*

- o **`ADL_PORT_EVENT_CTS_STATE_CHANGE`**

  *Informs the ADL application that the specified **Port** CTS signal state has changed to the new **State** value (0/1). This event will be received by all subscribers which have started a polling process on the specified **Port** CTS signal with the adl_portStartSignalPolling function.*
  *The handler **Port** parameter uses the **`adl_port_e`** type described above.*
  *The handler **State** parameter is set only for the ADL_PORT_EVENT_XXX_STATE_CHANGE events.*

- Returned values
  - o A positive or null handle on success ;
  - o **`ADL_RET_ERR_PARAM`** on parameter error,
  - o **`ADL_RET_ERR_NO_MORE_HANDLES`** if there is no more free handles (the service is able to process up 127 subscriptions).
  - o **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.23.5 The adl_portUnsubscribe Function

This function unsubscribes from the AT/FCM IO Ports service. The related handler will not receive ports related events any more. If a signal polling process was started only for this handle, it will be automaticaly stopped.

- Prototype

```
s8    adl_portUnsubscribe (u8       Handle );
```

- Parameters

  Handle:

  Handle previously returned by the **`adl_portSubscribe`** function.

- Returned values
  - o **`OK`** on success ;
  - o **`ADL_RET_ERR_UNKNOWN_HDL`** if the provided handle is unknown ;
  - o **`ADL_RET_ERR_NOT_SUBSCRIBED`** if the service is not subscribed.
  - o **`ADL_RET_ERR_SERVICE_LOCKED`** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.23.6 The adl_portIsAvailable Function

This function checks if the required port is currently opened or not.

- Prototype

```
bool  adl_portIsAvailable    ( adl_port_e Port );
```

- Parameters

  Port:

  Port from which to require the current state.

- Returned values
  - TRUE if the port is currently opened ;
  - FALSE if the port is closed, or if it does not exists.

  Notes
  - The function will always return TRUE on the **ADL_PORT_GSM_BASE** port ;
  - The function will always return TRUE on the **ADL_PORT_GPRS_BASE** port if the GPRS feature is enabled (always FALSE otherwise).

3.23.7 The adl_portGetSignalState Function

This function returns the required port signal state.

- Prototype

```
s8    adl_portGetSignalState (adl_port_e       Port,
                              adl_portSignal_e Signal );
```

- Parameters

  Port:

  Port from which to require the current signal state. Only physical output related ports (UARTX & USB ones, used as physical ports, or with the 27.010 protocol) may be used with this function.

  Signal:

  Signal from which to query the current state, based on the following type:

```
typedef enum
{
    ADL_PORT_SIGNAL_CTS,
    ADL_PORT_SIGNAL_DSR,
    ADL_PORT_SIGNAL_LAST
} adl_portSignal_e;
```

  Signals are detailed below:
  - **ADL_PORT_SIGNAL_CTS**

    Required port CTS input signal: physical pin in case of a physical port (UARTX), emulated logical signal in case of a 27.010 logical port.
  - **ADL_PORT_SIGNAL_DSR**

    Required port DSR input signal: physical pin in case of a physical port (UARTX), emulated logical signal in case of a 27.010 logical port.

- Returned values
  - o The signal state (0/1) on success ;
  - o **ADL_RET_ERR_PARAM** on parameter error;
  - o **ADL_RET_ERR_BAD_STATE** if the required port is not opened.

3.23.8 The adl_portStartSignalPolling Function

This function starts a polling process on a required port signal for the provided subscribed handle.

Only one polling process can run at a time. A polling process is defined on one port, for one or several of this port's signals.

It means that this function may be called several times on the same port in order to monitor several signals; the polling time interval is set up by the first function call (polling tme parameters are ignored or further calls). If the function is called several times on the same port & signal, additional calls will be ignored.

Once a polling process is started on a port's signal, this one is monitored: each time this signal state changes, a **ADL_PORT_EVENT_XXX_STATE_CHANGE** event is sent to all the handlers which have required a polling process on it.

Whatever is the number of requested signals and subscribers to this port polling process, a single cyclic timer will be internally used for this one.

- Prototype

```
s8    adl_portStartSignalPolling ( u8                Handle,
                                   adl_port_e        Port,
                                   adl_portSignal_e  Signal,
                                   u8                PollingTimerType,
                                   u32               PollingTimerValue );
```

- Parameters

Handle:

Handle previously returned by the **adl_portSubscribe** function.

Port:

Port on which to run the polling process. Only physical output related ports (UARTX & USB ones, used as physical ports, or with the 27.010 protocol) may be used with this function.

Signal:

Signal to monitor while the polling process. See the **adl_portGetSignalState** function for information about the available signals.

PollingTimerType:

PollingTimerValue parameter value's unit. The allowed values are defined below:

Timer type	Timer unit
ADL_TMR_TYPE_100MS	PollingTimerValue is in 100 ms steps
ADL_TMR_TYPE_TICK	PollingTimerValue is in 18.5 ms tick steps

This parameter is ignored on additional function calls on the same port.

PollingTimerValue:

Polling time interval (uses the PollingTimerType parameter's value unit).

This parameter is ignored on additional function calls on the same port.

- Returned values
  - **OK** on success ;
  - **ADL_RET_ERR_PARAM** on parameter error ;
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown ;
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if the service is not subscribed ;
  - **ADL_RET_ERR_BAD_STATE** if the required port is not opened ;
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if a polling process is already running on another port.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.23.9 The adl_portStopSignalPolling Function

This function stops a running polling process on a required port signal for the provided subscribed handle.

The associated handler will not receive the **ADL_PORT_EVENT_XXX_STATE_CHANGE** events related to this signal port anymore.

The internal polling process cyclic timer will be stopped as soon as the last subscriber to the current running polling process has call this function.

- Prototype

```
s8    adl_portStopSignalPolling (u8                Handle,
                                 adl_port_e        Port,
                                 adl_portSignal_e  Signal );
```

- Parameters

Handle:

Handle previously returned by the adl_portSubscribe function.

Port:

Port on which the polling process to stop is running.

Signal:

Signal on which the polling process to stop is running.

- Returned values
  - **OK** on success ;
  - **ADL_RET_ERR_PARAM** on parameter error ;
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown ;
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if the service is not subscribed ;
  - **ADL_RET_ERR_BAD_STATE** if the required port is not opened ;
  - **ADL_RET_ERR_BAD_HDL** if there is no running polling process for this Handle / Port / Signal combination.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.24 RTC Service

ADL provides a RTC service to access to the Wireless CPU®s inner RTC, and to process time related data.

The defined operations are:

- A **adl_rtcGetTime**
- A **adl_rtcSetTime**
- A **adl_rtcConvertTime**
- A **adl_rtcDiffTime**

3.24.1 Required Header File

The header file for the RTC functions is:

```
adl_rtc.h
```

3.24.2 RTC service Types

3.24.2.1 The adl_rtcTime_t Structure

Holds a RTC time:

```
typedef struct
{
    u32 Pad0                  // Not used
    u32 Pad1                  // Not used
    u16 Year;                 // Year (Four digits)
    u8  Month;                // Month (1-12)
    u8  Day;                  // Day of the Month (1-31)
    u8  WeekDay;              // Day of the Week (1-7)
    u8  Hour;                 // Hour (0-23)
    u8  Minute;               // Minute (0-59)
    u8  Second;               // Second (0-59)
    u32 SecondFracPart;       // Second fractional part
    u32 Pad2;                 // Not used
} adl_rtcTime_t;
```

Second fractional part (0-MAX) The MAX value is available from the registry field rtc_PreScalerMaxValue. See panel "Capabilities registry informations".

3.24.2.2 The adl_rtcTimeStamp_t Structure

Used to perform arithmetic operations on time data:

```
typedef struct
{
    u32 TimeStamp;           // Seconds elapsed since 1st January 1970
    u32 SecondFracPart;      // Second fractional part
} adl_rtcTimeStamp_t;
```

Second fractional part (0-MAX) The MAX value is available from the registry field rtc_PreScalerMaxValue. See panel "Capabilities registry informations".

3.24.2.3 Constants

RTC service constants are defined below:

Constant	Value	Use
ADL_RTC_DAY_SECONDS	24 * ADL_RTC_HOUR_SECONDS	*Seconds count in a day*
ADL_RTC_HOUR_SECONDS	60 * ADL_RTC_MINUTE_SECONDS	*Seconds count in an hour*
ADL_RTC_MINUTE_SECONDS	60	*Seconds count in a minute*
ADL_RTC_MS_US	1000	*µseconds count in a millisecond*

3.24.2.4 Macros

RTC service macros are defined below:

Macro	Parameter	Use
ADL_RTC_SECOND_FRACPART_STEP	adl_rtcGetSecondFracPartStep structure	*Second fractional part step value (in µs) extraction macro*
ADL_RTC_GET_TIMESTAMP_DAYS(_t)	(_t.TimeStamp / ADL_RTC_DAY_SECONDS) structure	*Days number extraction macro.*
ADL_RTC_GET_TIMESTAMP_HOURS(_t)	((_t.TimeStamp % ADL_RTC_DAY_SECONDS) / ADL_RTC_HOUR_SECONDS) structure	*Hours number extraction macro*
ADL_RTC_GET_TIMESTAMP_MINUTES(_t)	((_t.TimeStamp % ADL_RTC_HOUR_SECONDS) / ADL_RTC_MINUTE_SECONDS) structure	*Minutes number extraction macro*
ADL_RTC_GET_TIMESTAMP_SECONDS(_t)	(_t.TimeStamp % ADL_RTC_MINUTE_SECONDS) structure	*Seconds number extraction macro*
ADL_RTC_GET_TIMESTAMP_MS(_t)	(((u32)(_t.SecondFracPart * ADL_RTC_SECOND_FRACPAR T_STEP)) / ADL_RTC_MS_US) structure	*Milliseconds number extraction macro.*
ADL_RTC_GET_TIMESTAMP_US(_t)	(((u32)(_t.SecondFracPart * ADL_RTC_SECOND_FRACPAR T_STEP)) % ADL_RTC_MS_US) structure	*µseconds number extraction macro*

3.24.3 Enumerations

3.24.3.1 The adl_rtcConvert_e Type

This structure contains the available conversion modes.

- Code

```
typedef enum
{
        ADL_RTC_CONVERT_TO_TIMESTAMP,
        ADL_RTC_CONVERT_FROM_TIMESTAMP
} adl_rtcConvert_e;
```

- Description

ADL_RTC_CONVERT_TO_TIMESTAMP: Conversion mode to TimeStamp.

ADL_RTC_CONVERT_FROM_TIMESTAMP: Conversion mode from TimeStamp.

3.24.4 The adl_rtcGetSecondFracPartStep Function

This function retrieves the second fractional part step (in µs), reading the rtc_PreScalerMaxValue register field.

- Prototype

```
float adl_rtcGetSecondFracPartStep ( void );
```

- Returned values
  - The second fractional part step of the Wireless CPU®, in µs.

3.24.5 The adl_rtcGetTime Function

This function retrieves the current RTC time into an adl_rtcTime_t structure.

- Prototype

```
s32   adl_rtcGetTime ( adl_rtcTime_t * TimeStructure );
```

- Parameters

  TimeStructure:

  RTC structure where to copy current time.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** on parameter error.

3.24.6 The adl_rtcSetTime Function

This function sets a RTC time from a adl_rtcTime_t structure.

- Prototype

```
s32   adl_rtcSetTime ( adl_rtcTime_t * TimeStructure );
```

- Parameters

  TimeStructure:

  RTC structure where to get current time.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** on parameter error.

<u>Note:</u>

1: the input parameter cannot be a constant since it is modified by the API

2: when setting the RTC time SecondFracPart and WeekDay field are ignored.

3.24.7 The adl_rtcConvertTime Function

This function is able to convert RTC time structure to timestamp structure, and timestamp structure to RTC time structure thanks to a third agument precising the way of conversion.

- Prototype

```
s32   adl_rtcConvertTime (adl_rtcTime_t*       TimeStructure,
                          adl_rtcTimeStamp_t*  TimeStamp,
                          adl_rtcConvert_e     Conversion );
```

- Parameters

  TimeStructure:

  RTC structure where to get/set current time

  TimeStamp:

  Timestamp structure where to get/set current time

  Conversion:

  Conversion way:

  - ADL_RTC_CONVERT_TO_TIMESTAMP
  - ADL_RTC_CONVERT_FROM_TIMESTAMP

- Returned values
  - **OK** on success,
  - **ERROR** if conversion failed (internal error),
  - **ADL_RET_ERR_PARAM** on parameter error.

3.24.8 The adl_rtcDiffTime Function

This function reckons the difference between two timestamps.

- Prototype

```
s32   adl_rtcDiffTime ( adl_rtcTimeStamp_t *    TimeStamp1,
                        adl_rtcTimeStamp_t *    TimeStamp2,
                        adl_rtcTimeStamp_t *    Result );
```

- Parameters

  TimeStamp1:

  First timestamp to compare

  TimeStamp2:

  Second timestamp to compare

  Result:

  Reckoned time difference

- Returned values
  - **1** if TimeStamp1 is greater than TimeStamp2,
  - **-1** if TimeStamp2 is greater than TimeStamp1,
  - **0** if the provided TimeStamps are the same,
  - **ADL_RET_ERR_PARAM** on parameter error.

3.24.9 Capabilities

ADL provides informations to get the RTC Second Frac Part capabilities.

The following entry is defined in the registry:

Registry entry	Type	Description
rtc_PreScalerMaxValue	INTEGER	0: No second fractional part xxx: Second fractional part resolution

3.24.10 Example

This example demonstrates how to use the RTC service in a nominal case (error cases are not handled) with a Wireless CPU®.

Complete examples using the RTC service are also available on the SDK (generic Download library sample).

```
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Local variables
    adl_rtcTime_t Time1, Time2;
    adl_rtcTimeStamp_t Stamp1, Stamp2, DiffStamp;
    s32 Way;

    // Get time
    adl_rtcGetTime ( &Time1 );
    adl_rtcGetTime ( &Time2 );

    // Convert to time stamps
    adl_rtcConvertTime ( &Time1, &Stamp1, ADL_RTC_CONVERT_TO_TIMESTAMP );
    adl_rtcConvertTime ( &Time2, &Stamp2, ADL_RTC_CONVERT_TO_TIMESTAMP );

    // Reckon time difference
    Way = adl_rtcDiffTime ( &Stamp1, &Stamp2, &DiffStamp );

   //Convert the time difference from time stamps
   adl_rtcConvertTime (&Diff, &DiffStamp, ADL_RTC_CONVERT_FROM_TIMESTAMP );

   //Set back the initial time
   adl_rtcSetTime ( &Time1 );
}
```

3.25 IRQ Service

The ADL IRQ service allows interrupt handlers to be defined.

These handlers are usable with other services (External Interrupt Pins, Audio) to monitor specific interrupt sources.

Interrupt handlers are running in specific execution contexts of the application. Please refer to the Execution Contexts Service for more information (§ 3.27).

The defined operations are:

- Subscription functions **adl_irqSubscribe** & **adl_irqSubscribeExt** to define interrupt handlers
- Configuration functions **adl_irqSetConfig** & **adl_irqGetConfig** to handle interrupt handlers configuration
- An Unsubscription function **adl_irqUnsubscribe** to remove an IRQ handler definition
- A Get Capabilities function **adl_irqGetCapabilities** to retrieve the IRQ service capabilities

Note:

The Real Time Enhancement feature has to be enabled on the Wireless CPU® in order to make this service available.

The Real Time Enhancement feature state can be read thanks to the AT+WCFM=5 command response value: this feature state is represented by the bit 4 (00000010 in hexadecimal format)

Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU®.

3.25.1 Required Header File

The header file for the IRQ functions is:

adl_irq.h

3.25.2 The adl_irqID_e Type

This type defines the interrupt sources that the service is able to monitor.

```
typedef enum
{
  ADL_IRQ_ID_AUDIO_RX_LISTEN,
  ADL_IRQ_ID_AUDIO_TX_LISTEN,
  ADL_IRQ_ID_AUDIO_RX_PLAY,
  ADL_IRQ_ID_AUDIO_TX_PLAY,
  ADL_IRQ_ID_EXTINT,
  ADL_IRQ_ID_TIMER,
  ADL_IRQ_ID_EVENT_CAPTURE
  ADL_IRQ_ID_EVENT_DETECTION
  ADL_IRQ_ID_SPI_EOT,
  ADL_IRQ_ID_I2C_EOT,
  ADL_IRQ_ID_LAST       // Reserved for internal use
} adl_irqID_e;
```

The **ADL_IRQ_ID_AUDIO_RX_LISTEN** constant identifies RX path interrupt sources raised by the Audio Stream Listen service. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_AUDIO_TX_LISTEN** constant identifies TX path interrupt sources raised by the Audio Stream Listen service. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_AUDIO_RX_PLAY** constant identifies RX path interrupt sources raised by the Audio Stream Play service. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_AUDIO_TX_PLAY** constant identifies TX path interrupt sources raised by the Audio Stream Play. Please refer to the Audio Service for more information.

The **ADL_IRQ_ID_EXTINT** constant identifies interrupt sources raised by the External Interrupt Pin source. For more information, please refer to the ExtInt service (see section 3.27.)

The **ADL_IRQ_ID_TIMER** constant identifies interrupt sources raised by the Timer Interrupts source. For more information, please refer to the TCU service (see section 3.26).

The **ADL_IRQ_ID_EVENT_CAPTURE** constant identifies capture interrupt sources raised by the Timer Interrupts source. For more information, please refer to the TCU service (see section 3.26).

The **ADL_IRQ_ID_EVENT_DETECTION** constant identifies detection interrupt sources raised by the Timer Interrupt source. For more information, please refer to the TCU service (see section 3.26).

The **ADL_IRQ_ID_SPI_EOT** constant identifies SPI bus asynchronous end of transmission event. Please refer to the BUS service (see section 3.11) for more information.

The **ADL_IRQ_ID_I2C_EOT** constant identifies I2C bus asynchronous end of transmission event. Please refer to the BUS service (see section 3.11) for more information.

3.25.3 The adl_irqNotificationLevel_e Type

This type defines the notification level of a given interrupt handler.

For more information on specific high and low level handlers behavior, please refer to the Execution Context Service description (§ 3.28).

```
typedef enum
{
  ADL_IRQ_NOTIFY_LOW_LEVEL,
  ADL_IRQ_NOTIFY_HIGH_LEVEL,
  ADL_IRQ_NOTIFY_LAST          // Reserved for internal use
} adl_irqNotificationLevel_e;
```

The **ADL_IRQ_NOTIFY_LOW_LEVEL** constant allows low level interrupt handlers to be defined.

The **ADL_IRQ_NOTIFY_HIGH_LEVEL** constant allows high level interrupt handlers to be defined.

3.25.4 The adl_irqPriorityLevel_e Type

This type defines the priority level of a given interrupt handler.

The lowest priority level is always 0.

The highest priority level shall be retrieved thanks to the **adl_irqGetCapabilities** function.

Please refer to each interrupt related service for more information about the available priority levels.

The priority level of a handler allows the notification order to be set in case of event conflict:

- A N priority level handler cannot be interrupted by other handlers with the same N priority level, or with a lower N - X priority level.
- A N priority level handler can be interrupted by any other handlers with an higher N + X priority level.

Note:

Priority levels settings are significant only for low level interrupt handlers. There is no way to define priority levels for high level interrupt handlers.

Priority levels settings are only efficient with external interrupt service, allowing to configure the several external interrupt pins priority. Other interrupt source services priorities are not configurable, and always have the values listed in the table below. Trying to modify the priority of such services will have no effect.

Service	Events	Priority value
Audio Service	**ADL_IRQ_ID_AUDIO_RX_LISTEN** **ADL_IRQ_ID_AUDIO_TX_LISTEN** **ADL_IRQ_ID_AUDIO_RX_PLAY** **ADL_IRQ_ID_AUDIO_TX_PLAY**	Max
BUS & TCU Services	**ADL_IRQ_ID_SPI_EOT** **ADL_IRQ_ID_I2C_EOT** **ADL_IRQ_ID_TIMER** **ADL_IRQ_ID_EVENT_CAPTURE** **ADL_IRQ_ID_EVENT_DETECTION**	0

MAX value represents the maximum priority value.

3.25.5 The adl_irqEventData_t Structure

This structure supplies interrupt handlers with data related to the interrupt source.

```
typedef struct
{
  union
  {
    void * LowLevelOuput;
    void * HighLevelInput;
  } UserData;
    void * SourceData;
    u32    Instance
    void * Context
} adl_irqEventData_t;
```

3.25.5.1 The UserData Field

This field allows the application to exchange data between low level and high level interrupt handlers.

3.25.5.2 The Source Data Field

This field provides to handlers an interrupt source specific data. Please refer to each interrupt source related service for more information about this field data structure.

When the interrupt occurs, the source related information structure is automatically provided by the service to the low level interrupt handler, whatever if the **ADL_IRQ_OPTION_AUTO_READ** option is enabled or not. In an high level interrupt handler, this field will be set only if the **ADL_IRQ_OPTION_AUTO_READ** option is enabled.

3.25.5.3 The Instance Field

Instance identifier of the interrupt event which has just occurred. Please refer to each interrupt source related service for more information on the instance number use.

3.25.5.4 The Context Field

Application context, given back by ADL on event occurrence. This context was provided by the application to the interrupt source related service, when using the operation which enables the interrupt event occurrences. If the interrupt source related service does not offer a way to define an application context, this member will be set to NULL. Please refer to each interrupt source related service for more information on the instance number use.

3.25.6 The adl_irqCapabilities_t Structure

This structure allows the application to retrieve information about the IRQ service capabilities.

```
typedef struct
{
  u8  PriorityLevelsCount,
  u8  Pad [3]                    // Reserved for internal use
  u8  InstancesCount [ADL_IRQ_ID_LAST]
} adl_irqCapabilities_t;
```

3.25.6.1 The PriorityLevelsCount Field

This field provides the priority levels count, usable to set an **adl_irqPriorityLevel_e** type value (see section 3.25.4)

Such a value shall use a range from 0 to **PriorityLevelsCount**-1.

3.25.6.2 The InstancesCount Field

This field provides the instances count, for each interrupt source identifier. Please refer to each interrupt source related service for more information. If an instance count value is set to 0, the corresponding interrupt related event is not supported on the current platform.

3.25.7 The adl_irqConfig_t Structure

This structure allows the application to configure interrupt handlers behaviour.

```
typedef struct
{
  adl_irqPriorityLevel_e    PriorityLevel,
  bool                      Enable,
  u8                        Pad[2]      // Reserved for future use
  adl_irqOptions_e          Options
} adl_irqConfig_t;
```

3.25.7.1 The PriorityLevel Field

This field defines the interrupt handler priority level. Please refer to the **adl_irqPriorityLevel_e** type definition for more information (see § 3.25.4).

Note:

If different services are plugged on an interrupt handler, the priority value will be applied to all services, if possible. If the priority value is not applicable for a given service, it will be ignored.

3.25.7.2 The Enable Field

This field defines if the interrupt handler is enabled or not.
If set to **TRUE**, the interrupt handler is enabled and any interrupt event on which is plugged this handler will call the related function.
If set to **FALSE**, the interrupt handler is disabled: all interrupt events on which are plugged this handler are masked, and will be delayed until the handler is enabled again.

Note:

This is the default behaviour. If specified in the related service, the event shall be just delayed until the handler is enabled again.

3.25.7.3 The Options Field

This field defines the interrupt handler notification options. A bitwise OR combination of the option constants has to be used. Please refer to the 3.25.8 **adl_irqOptions_e** type definition for more information.

3.25.8 The adl_irqOptions_e type

These options have to be used with a bit-wise OR in order to specify the interrupt handler behaviour.

```
typedef enum
{
   ADL_IRQ_OPTION_AUTO_READ                  =1UL,
   ADL_IRQ_OPTION_PRE_ACKNOWLEDGEMENT        =0UL,
   ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT       =0UL
} adl_ adl_irqOptions_e;
```

ADL_IRQ_OPTION_AUTO_READ: Automatic interrupt source information read.

When the interrupt occurs, the source related information structure is automatically read by the service, and supplied to the low level interrupt handler.
When used with a high level interrupt handler, this option allows the application to get the source related information structure read at interrupt time.

Note:

This option has no effect with a low level interrupt handler (**adl_irqEventData_t::SourceData** field will always be provided by the related interrupt service in this case).

ADL_IRQ_OPTION_PRE_ACKNOWLEDGEMENT: Interrupt source pre-acknowledgement.

ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT: Interrupt source post-acknowledgement.

3.25.9 The adl_irqHandler_f Type

This type has to be used by the application in order to provide ADL with an interrupt hander.

- Prototype

```
typedef bool (*)adl_irqHandler_f (adl_irqID_e         Source,
                                  adl_irqNotificationLevel_e
                                           NotificationLevel,
                                  adl_irqEventData_t * Data );
```

- Parameter

  Source:

  Interrupt source identifier.

  Please refer to **adl_irqID_e** type definition for more information. (see § 3.25.2).

  NotificationLevel:

  Interrupt handler current notification level.

  Please refer to **adl_irqNotifyLevel_e** type definition for more information (see § 3.25.3).

  Data:

  Interrupt handler input/output data field.

  Please refer to **adl_irqEventData_e** type definition for more information. (see § 3.25.5).

- Returned values
  - o Not relevant for high level interrupt handlers.
  - o For low level interrupt handlers
    - TRUE: requires ADL to call the subscribed high level handler for this interrupt source.
    - FALSE: requires ADL not to call any high level handler for this interrupt source.

  Note:

  For low level interrupt handlers, 1 ms can be considered as a maximum latency time before being notified with the interrupt source event.

3.25.10 The adl_irqSubscribe Function

This function allows the application to supply an interrupt handler, to be used later in Interrupt source related service subscription.

- Prototype

```
s32 adl_irqSubscribe ( adl_irqHandler_f              IrqHandler,
                       adl_irqNotificationLevel_e    NotificationLevel,
                       adl_irqPriorityLevel_e        PriorityLevel,
                       adl_irqOptions_e              Options );
```

- Parameter

IrqHandler:

Interrupt handler supplied by the application.

NotificationLevel:

Interrupt handler notification level; allows the supplied handler to be identified as a low level or a high level one.

PriorityLevel:

Interrupt handler priority level; Please refer to **adl_irqPriorityLevel_e** type definition for more information (see § 3.25.4).

Options:

Interrupt handler notification options.

A bitwise OR combination of the options constant has to be used. Please refer to the **adl_irqOptions_e** type definition for more information (see section 3.25.8).

- Returned values
  - o Handle: A positive or null IRQ service handle on success, to be used in further IRQ & interrupt source services function calls.
  - o **ADL_RET_ERR_PARAM** on a supplied parameter error.
  - o **ADL_RET_ERR_NOT_SUBSCRIBED** if a low or high level handler subscription is required while the associated context call stack size was not supplied by the application (please refer to the Mandatory Service description (§ 3.1)).
  - o **ADL_RET_ERR_BAD_STATE** if the function is called in RTE mode.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

<u>Note:</u>

- o The IRQ service will always return an error code in RTE mode (the service is not supported in this mode). .Use of the IRQ service should be flagged in order to make an application working correctly in RTE.
- o This function is a shortcut to the **adl_irqSubscribeExt** one. Provided **PriorityLevel** and **Options** parameters values will be used to fill the configuration structure. The **adl_irqConfig_t::Enable** field will be set to TRUE by default.

3.25.11 The adl_irqSubscribeExt Function

This function allows the application to supply an interrupt handler, to be used later in Interrupt source related service subscription.

- Prototype

```
s32 adl_irqSubscribeExt (adl_irqHandler_f             IrqHandler,
                          adl_irqNotificationLevel_e  NotificationLevel,
                          adl_irqConfig_t*            Config );
```

- Parameter

IrqHandler:

Interrupt handler supplied by the application.

Please refer to **adl_irqHandler_f** type definition for more information (see § 3.25.9).

NotificationLevel:

Interrupt handler notification level; allows the supplied handler to be identified as a low level or a high level one.

Please refer to **adl_irqNotifyLevel_e** type definition for more information (see § 3.25.3).

Config:

Interrupt handler configuration. Please refer to the 3.25.7 **adl_irqConfig_t** structure definition for more information.

- Returned values
  - Handle: A positive or null IRQ service handle on success, to be used in further IRQ & interrupt source services function calls.
  - **ADL_RET_ERR_PARAM** on a supplied parameter error.
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if a low or high level handler subscription is required while the associated context call stack size was not supplied by the application (please refer to the Mandatory Service description (§ 3.1)).
  - **ADL_RET_ERR_BAD_STATE** if the function is called in RTE mode.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note:

The IRQ service will always return an error code in RTE mode (the service is not supported in this mode). Use of the IRQ service should be flagged in order to make an application working correctly in RTE.

3.25.12 The adl_irqUnsubscribe Function

This function allows the application to unsubscribe from the interrupt service. The associated handler will no longer be notified of interrupt events.

- Prototype

```
s32 adl_irqUnsubscribe ( s32   IrqHandle );
```

- Parameter

  IrqHandle:

  Interrupt service handle, previously returned by the **adl_irqSubscribe** function.

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the supplied handle is unknown.
  - o **ADL_RET_ERR_BAD_STATE** if the supplied handle is still used by an interrupt source service.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.25.13 The adl_irqSetConfig function

This function allows the application to update an interrupt handler's configuration.

- Prototype

```
s32 adl_irqSetConfig  ( s32                     IrqHandle,
                        adl_irqConfig_t *       Config )
```

- Parameter

  IrqHandle:

  IRQ service handle, previously returned by the **adl_irqSubscribe** function.

  Config:

  Interrupt handler configuration structure. Please refer to the **adl_irqConfig_t** structure definition for more information (see section 3.25.7).

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the supplied handle is unknown.
  - o **ADL_RET_ERR_PARAM** on a supplied parameter error.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.25.14 The adl_irqGetConfig function

This function allows the application to retrieve an interrupt handler's configuration.

- Prototype

```
s32 adl_irqGetConfig  ( s32                     IrqHandle,
                        adl_irqConfig_t *       Config )
```

- Parameter

  IrqHandle:

  IRQ service handle, previously returned by the **adl_irqSubscribe** function.

  Config:

  Interrupt handler configuration structure. Please refer to the **adl_irqConfig_t** structure definition for more information (see section 3.25.7).

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied handle is unknown.
  - **ADL_RET_ERR_PARAM** on a supplied parameter error.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.25.15 The adl_irqGetCapabilities Function

This function allows the application to retrieve information about the IRQ service capabilities on the current platform.

- Prototype

```
s32 adl_irqGetCapabilities  ( adl_irqCapabilities_t *  Capabilities )
```

- Parameter

  Capabilities

  IRQ service capabilities information structure. Please refer to the **adl_irqCapabilities_t** structure definition for more information (see section 3.25.6).

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** on parameter error.

3.25.16 Example

The code sample below illustrates a nominal use case of the ADL IRQ Service public interface (error cases are not handled).

```
// Global variable: IRQ service handle
  s32 MyIRQHandle;

  // Interrupt handler
  bool MyIRQHandler ( adl_irqID_e Source, adl_irqNotificationLevel_e
  NotificationLevel, adl_irqEventData_t * Data )
  {
      // Interrupt process...
      // Notify the High Level handler, if any
      return TRUE;
  }

  // Somewhere in the application code, used as event handler
  void MyFunction1 ( void )
  {
      // Local variables
      adl_irqCapabilities_t Caps;
      adl_irqConfig_t Config;

      // Get capabilities
      adl_irqGetCapabilities ( &Caps );

      // Set configuration
      Config.PriorityLevel = Caps.PriorityLevelsCount - 1; // Highest
priority
      Config.Enable = TRUE;              // Interrupt handler enabled
      Config.Options = ADL_IRQ_OPTION_AUTO_READ;          // Auto-read
option set

      // Subscribe to the IRQ service
      MyIRQHandle = adl_irqSubscribeExt ( MyIRQHandler,
      ADL_IRQ_NOTIFY_LOW_LEVEL, &Config );

      // TODO: Interrupt source service subscription
      ...

      // Mask the interrupt
      adl_irqGetConfig ( MyIRQHandle, &Config );
      Config.Enable = FALSE;
      adl_irqSetConfig ( MyIRQHandle, &Config );

      ...
```

```
    // Unmask the interrupt
    adl_irqGetConfig ( MyIRQHandle, &Config );
    Config.Enable = TRUE;
    adl_irqSetConfig ( MyIRQHandle, &Config );

    ...

    // TODO: Interrupt source service un-subscription
    ...

    // Un-subscribe from the IRQ service
    adl_irqUnsubscribe ( MyIRQHandle );
}
```

3.26 TCU Service

ADL supplies Timer & Capture Unit Service interface to handle operations related to the Wireless CPU® hardware timers & capture units.

The defined operations are:

- A subscription function (**adl_tcuSubscribe**) to subscribe to the TCU service
- An unsubscription function (**adl_tcuUnsubscribe**) to unsubscribe from the TCU service
- Start & Stop functions (**adl_tcuStart** & **adl_tcuStop**) to control the TCU service event generation

3.26.1 Required Header File

The header file for the TCU function is:

adl_tcu.h

3.26.2 Capabilities Registry Informations

ADL provides capabilities information about the TCU service, thanks to the registry service.

The following entries have been defined in the registry:

Registry entry	Type	Description
tcu_TmrSrvAvailable	INTEGER	Availability of the Accurate Timer service (boolean value)
tcu_CaptSrvAvailable	INTEGER	Availability of the Event Capture service (boolean value)
tcu_DetectSrvAvailable	INTEGER	Availability of the Event Detection service (boolean value)
tcu_EvPinsNb	INTEGER	Number of pins usable to monitor events with the Capture & Detection services
tcu_TimersNb	INTEGER	Maximum number of Accurate Timer service instances which can be running at the same time
tcu_TimerBoundaries	DATA	Minimum & maximum duration values which can be used for the Accurate Timer service, using the **adl_tcuTimerBoundaries_t** structure format.

Registry entry	Type	Description
tcu_EvDetectUnit	INTEGER	Time unit used (following adl_tcuTimerUnit_e type) in the event detection service: • for inactivity period settings (_adl_tcuEventDetectionSettings_t::Duration) • for last stable state duration information (_adl_tcuEventDetectionInfo_t::LastStateDuration)
tcu_EvCaptUnit	INTEGER	Time unit used (following **adl_tcuTimerUnit_e** type) in the event capture service, for capture duration setting (**_adl_tcuEventCaptureSettings_t::Duration**)

3.26.3 Data Structures

3.26.3.1 The adl_tcuEventCaptureSettings_t Structure

TCU configuration structure, when the **ADL_TCU_EVENT_CAPTURE** service is used.

```
typedef struct
{
     u16                     CapturePinID
    adl_tcuEventType_e       EventType
    u32                      Duration
    u32*                     EventCounter
} adl_tcuEventCaptureSettings_t;
```

- Fields

  CapturePinID:

  Identifier of the pin on which the service has to monitor events.Please refer to the PTS for more information. The allowed values range is from 0 to the value returned by the tcu_EvPinsNb capability - 1.

  EventType:

  Event capture type, using one of the **adl_tcuEventType_e** type values.

  Duration:

  Duration of the capture period (in the unit provided by the **tcu_EvCaptUnit** capability). This duration is used only if the **adl_tcuEventCaptureSettings_t::EventCounter** address is set to **NULL**, otherwise it will be ignored. When the parameter is used, the related IRQ service handlers are called on each duration expiration, indicating to the application how many events have occurred since the previous handler call.

Note: When the Event Capture is configured with a period duration greater than 0, an Accurate Timer resource is internally used to handle the service.

EventCounter:

Address of a 32 bits variable provided by the application, where the events counter value has to be stored. If this address is provided, no interrupt events will be generated, but the event counter value will be incremented each time a new event is detected. Please note that in this case, none of IRQ service handles provided to the **adl_tcuSubscribe** function will be used (parameters values will be ignored). If this address is set to **NULL**, the service will regularly generate events, on the time base defined by the **adl_tcuEventCaptureSettings_t::Duration** parameter.

Note: The provided variable address has to be accessible from the Firmware until the service is unsubscribed. This means that the variable has to be either a global/static one, or an allocated heap buffer.

If provided, the event counter content is only incremented, but never reset by the TCU service. The application has to reset it by itself when it is necessary.

3.26.3.2 The adl_tcuEventDetectionInfo_t Structure

This structure contains the information provided to event handlers when **ADL_IRQ_ID_EVENT_DETECTION** events are generated, following a **ADL_TCU_EVENT_CAPTURE** service subscription.

```
typedef struct
{
    u32                    LastStateDuration
   adl_tcuEventType_e      EventType
} adl_tcuEventDetectionInfo_t;
```

- Fields

LastStateDuration:

Duration (in the unit provided by the **tcu_EvDetectUnit** capability) of the last stable state of the monitored signal, before the handler notification occured.

EventType:

Type of the event which has caused the notification. If the value is positive or null, it represents the detected event type, using the **adl_tcuEventType_e** enumeration type. If the value is **ADL_TCU_EVENT_TYPE_NONE**, it means that no event has been detected since the last handler notification when the timeout programed thanks to the **adl_tcuEventDetectionSettings_t::Duration** parameter has elapsed.

3.26.3.3 The adl_tcuEventDetectionSettings_t Structure

TCU configuration structure, when the **ADL_TCU_EVENT_DETECTION** service is used.

```
typedef struct
{
        u16                         DetectionPinID
        adl_tcuEventType_e          EventType
        u32                         Duration
} adl_tcuEventDetectionSettings_t;
```

- Fields

DetectionPinID

Identifier of the pin on which the service has to monitor events. Please refer to the Product Technical Specification for more information. The allowed values range is from 0 to the value returned by the **tcu_EvPinsNb** capability - 1.

EventType

Event detection type, using one of the **adl_tcuEventType_e** type values.

Duration

Optional inactivity detection period duration, used to cause an handler notification if no event occurred for a given time slot. If this value is set to 0, the inactivity detection will be disabled. If this value is greater than 0, it is the inactivity detection period duration (in the unit provided by the **tcu_EvDetectUnit** capability): if no event has occurred since the last notification (or since the **adl_tcuStart** function call) when the duration expires, the associated handlers will be called to warn the application about this inactivity.

Note: When the Event Detection is configured with an inactivity period duration greater than 0, an Accurate Timer resource is internally used to handle the service.

3.26.3.4 The adl_tcuTimerBoundaries_t Structure

This structure is usable to retrieve the TCU capabilities about the Accurate Timer service duration boundaries.

```
typedef struct
{
        adl_tcuTimerDuration_t      MinDuration
        adl_tcuTimerDuration_t      MaxDuration
} adl_tcuEventDetectionInfo_t;
```

- Fields

MinDuration

Minimum timer duration, using the **adl_tcuTimerDuration_t** structure.

MaxDuration

Maximum timer duration, using the **adl_tcuTimerDuration_t** structure.

3.26.3.5 The adl_tcuTimerDuration_t Structure

Configuration structure usable to represent a timer duration.

```
typedef struct
{
    u32                     DurationValue
    adl_tcuTimerUnit_e      DurationUnit
} adl_tcuTimerDuration_t;
```

- Fields

DurationValue

Timer duration value, in the unit set by the **_adl_tcuTimerDuration_t::DurationUnit** field.

DurationUnit

Timer duration unit, using one of the **adl_tcuTimerUnit_e** type values.

3.26.3.6 The adl_tcuTimerSettings_t Structure

TCU configuration structure, when the **ADL_TCU_ACCURATE_TIMER** service is used.

```
typedef struct
{
    adl_tcuTimerDuration_t  Duration
    u32                     Periodic
} adl_tcuTimerSettings_t;
```

- Fields

Duration

Timer duration, using the **adl_tcuTimerDuration_t** configuration structure.

Periodic

Boolean periodic timer configuration:

if set to **TRUE**, the timer is reloaded after each event occurrence.

Otherwise, the timer is stopped after the first event occurrence.

3.26.4 Enumerators

3.26.4.1 The adl_tcuService_e Type

This enumeration lists the available TCU services types.

- code

```
enum
{ ADL_TCU_ACCURATE_TIMER,
  ADL_TCU_EVENT_CAPTURE,
  ADL_TCU_EVENT_DETECTION
 } adl_tcuService_e;
```

- description

ADL_TCU_ACCURATE_TIMER

Accurate timer service

Allows the application to subscribe to the accurate timer service.

Please refer to the Accurate Timers service configuration for more information.

ADL_TCU_EVENT_CAPTURE

Event capture service.

Allows the application to subscribe to the event capture service.

Please refer to the Event Capture service configuration for more information

ADL_TCU_EVENT_DETECTION

Event detection service.

Allows the application to subscribe to the event detection service.

Please refer to the Event Detection service configuration for more information.

3.26.4.2 The adl_tcuEventType_e Type

This enumeration lists the available event types usable for the capture & detection services.

- code

```
enum
{ ADL_TCU_EVENT_TYPE_NONE         = (s16)0xFFFF,  // No event detected
  ADL_TCU_EVENT_TYPE_RISING_EDGE = 0,              // Capture or detect
                                                   rising edge events only
  ADL_TCU_EVENT_TYPE_FALLING_EDGE,                 // Capture or detect
                                                   falling edge events only
  ADL_TCU_EVENT_TYPE_BOTH_EDGE                     // Capture or detect
                                                  events on both edges
} adl_tcuEventType_e;
```

Note:

ADL_TCU_EVENT_TYPE_NONE is only used for event detection information, as a **_adl_tcuEventDetectionInfo_t::EventType parameter** value.

3.26.5 Accurate Timers Service

This service is usable to generate (periodically or not) accurate timer events, configured thanks to the **adl_tcuTimerSettings_t** structure (such a structure has to provided to the **adl_tcuSubscribe** function).

Output parameter of the **adl_tcuStop** function is used as an **adl_tcuTimerDuration_t** pointer to return the remaining time until the timer expiration when the stop operation has been performed.

Interrupt handlers defined in the IRQ service - using the **adl_irqHandler_f** type - and provided at subscription time will be notified with the following parameters, according to the service configuration, and as soon as the **adl_tcuStart** function is called:

- the Source parameter will be set to **ADL_IRQ_ID_TIMER**
- the **adl_irqEventData_t::SourceData** field of the Data parameter will be set to **NULL**.
- the **adl_irqEventData_t::Instance** field of the Data parameter will be set to 0.
- the **adl_irqEventData_t::Context** field of the Data parameter will be set to the application context, provided at subscription time.

3.26.5.1 The adl_tcuTimerUnit_e Type

This enumeration lists the available duration units for the timer service.

```
typedef enum
{
      ADL_TCU_TIMER_UNIT_US          = 1,
      ADL_TCU_TIMER_UNIT_MS          = 1000,
      ADL_TCU_TIMER_UNIT_S           = 100000,
      ADL_TCU_TIMER_UNIT_ALIGN       = 0x7FFFFFFF
} adl_tcuTimerUnit_e;
```

- Description

ADL_TCU_TIMER_UNIT_US:	Timer duration is in microseconds.
ADL_TCU_TIMER_UNIT_MS:	Timer duration is in milliseconds.
ADL_TCU_TIMER_UNIT_S:	Timer duration is in seconds.
ADL_TCU_TIMER_UNIT_ALIGN	Reserved for internal use.

3.26.5.2 Example

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **ADL_TCU_ACCURATE_TIMER** mode.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// IRQ service handle
s32 IrqHandle;

// TCU Accurate timer configuration: periodic 5ms timer
adl_tcuTimerSettings_t Config = { { 5, ADL_TCU_TIMER_UNIT_MS }, TRUE };

// TCU interrupt handler
bool MyTCUHandler (adl_irqID_e Source, adl_irqNotificationLevel_e
NotificationLevel, adl_irqEventData_t * Data );
{
    // Check for Timer event
    if ( Source == ADL_IRQ_ID_TIMER )
    {
        // Trace event
        TRACE (( 1, "Timer event" ));
    }
      return TRUE;
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribes to the IRQ service
    IrqHandle = adl_irqSubscribe ( MyTCUHandler, ADL_IRQ_NOTIFY_LOW_LEVEL,
0, 0 );

    // Subscribes to the TCU service, in Accurate Timer mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_ACCURATE_TIMER, IrqHandle, 0,
&Config, NULL );

    // Starts event generation
    adl_tcuStart ( TCUHandle );
}
void MyFunction2 ( void )
{
    // Stops event generation, and gets remaining time
    u32 RemainingTimer
    adl_tcuStop ( TCUHandle, &RemainingTimer );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.6 Event Capture Service

This service is usable to count events on a given Wireless CPU® pin, and is configured thanks to the **adl_tcuEventCaptureSettings_t** structure (such a structure has to provided to the **adl_tcuSubscribe** function).

Output parameter of the **adl_tcuStop** function is not used for this service, and shall be set to **NULL.**

Interrupt handlers defined in the IRQ service - using the **adl_irqHandler_f** type - and provided at subscription time will be notified with the following parameters, according to the service configuration, and as soon as the **adl_tcuStart** function is called:

- the Source parameter will be set to **ADL_IRQ_ID_EVENT_CAPTURE**
- the **adl_irqEventData_t::SourceData** field of the Data parameter will have to be casted as an u32 value, indicating the number of events which have occured since the last event handler call. The notification period is configured by the **adl_tcuEventCaptureSettings_t::Duration** parameter.
- the **adl_irqEventData_t::Instance** field of the Data parameter will be set to the monitored pin identifier, required at subscription time in the **adl_tcuEventCaptureSettings_t::CapturePinID**.
- the **adl_irqEventData_t::Context** field of the Data parameter will be set to the application context, provided at subscription time.

3.26.6.1 Example (without handler notification)

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **ADL_TCU_EVENT_CAPTURE** mode, without handler notification.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// Event counter to be provided to the API
u32 MyEventCounter;

// TCU Event capture configuration: on pin 0, count falling edges, with a
provided event counter
adl_tcuEventCaptureSettings_t Config = { 0, ADL_TCU_EVENT_TYPE_FALLING_EDGE,
0, &MyEventCounter };

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{

    // Subscribes to the TCU service, in Event Capture mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_EVENT_CAPTURE, 0, 0, &Config,
NULL );

    // Reset counter to 0, and starts event generation
    MyEventCounter = 0;
    adl_tcuStart ( TCUHandle );
}

void MyFunction2 ( void )
{
    // Periodically monitor the events counter, whenever in the
application's life
    TRACE (( 1, "Current events count: %d", MyEventCounter ));
}

void MyFunction3 ( void )
{
    // Stops event generation
    adl_tcuStop ( TCUHandle, NULL );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.6.2 Example (with handler notification)

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **ADL_TCU_EVENT_CAPTURE** mode, with handler notification.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// IRQ service handle
s32 IrqHandle;

// TCU Event capture configuration: on pin 0, counts rising edge events, and
notify the handler every second
adl_tcuEventCaptureSettings_t Config = { 0, ADL_TCU_EVENT_TYPE_RISING_EDGE,
8, NULL };

// TCU interrupt handler
bool MyTCUHandler (adl_irqID_e Source, adl_irqNotificationLevel_e
NotificationLevel, adl_irqEventData_t * Data );
{
    // Check for Event Capture
    if ( Source == ADL_IRQ_ID_EVENT_CAPTURE )
    {
        // Check for pin identifier
        if ( Data->Instance == 0 )
        {
            // Get Source Data
            u32 SourceData = ( u32 ) Data->SourceData;

            // Trace event count
            TRACE (( 1, "%d events capture since last notification",
SourceData ));
        }
    }

    return TRUE;
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{

    // Subscribes to the IRQ service
    IrqHandle = adl_irqSubscribe ( MyTCUHandler, ADL_IRQ_NOTIFY_LOW_LEVEL,
0, ADL_IRQ_OPTION_AUTO_READ );

    // Subscribes to the TCU service, in Event Capture mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_EVENT_CAPTURE, IrqHandle, 0,
&Config, NULL );

    // Starts event generation
    adl_tcuStart ( TCUHandle );
```

```
}

void MyFunction2 ( void )
{
    // Stops event generation
    adl_tcuStop ( TCUHandle, NULL );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.7 Event Detection Service

This service is usable to detect events on a given Wireless CPU® pin, and is configured thanks to the **adl_tcuEventDetectionSettings_t** structure (such a structure has to provided to the **adl_tcuSubscribe** function.

Output parameter of the **adl_tcuStop** function is not used for this service, and shall be set to **NULL.**

Interrupt handlers defined in the IRQ service - using the **adl_irqHandler_f** type - and provided at subscription time will be notified with the following parameters, according to the service configuration, and as soon as the **adl_tcuStart** function is called.

- the Source parameter will be set to **ADL_IRQ_ID_EVENT_DETECTION**
- the **adl_irqEventData_t::SourceData** field of the Data parameter will have to be casted as a pointer on an **adl_tcuEventDetectionInfo_t** structure.
- the **adl_irqEventData_t::Instance** field of the Data parameter will be set to the monitored pin identifier, required at subscription time in the **adl_tcuEventDetectionSettings_t::DetectionPinID.**
- the **adl_irqEventData_t::Context** field of the Data parameter will be set to the application context, provided at subscription time.

3.26.7.1 Example

The code sample below illustrates a nominal use case of the ADL Timer & Capture Unit Service, in **ADL_TCU_EVENT_DETECTION** mode.

```
// Global variables

// TCU service handle
s32 TCUHandle;

// IRQ service handle
s32 IrqHandle;

// TCU Event detection configuration: on pin 0, detects rising edge events,
and set a 200 ms timeout
adl_tcuEventDetectionSettings_t Config = { 0,
ADL_TCU_EVENT_TYPE_RISING_EDGE, 200 };

// TCU interrupt handler
bool MyTCUHandler (adl_irqID_e Source, adl_irqNotificationLevel_e
NotificationLevel, adl_irqEventData_t * Data );
{
    // Check for Event Detection
    if ( Source == ADL_IRQ_ID_EVENT_DETECTION )
    {
        // Check for pin identifier
        if ( Data->Instance == 0 )
        {
            // Get Source Data
            adl_tcuEventDetectionInfo_t * SourceData =
           ( adl_tcuEventDetectionInfo_t * ) Data->SourceData;

            // Check for true or inactivity event
            if ( SourceData->EventType < 0 )
            {
                // Trace inactivity
                TRACE (( 1, "Event detection timeout" ));
            }
            else
            {
                // Trace event detection
                TRACE (( 1, "%d event detected; last state duration: %d ms",
                SourceData->EventType, SourceData->LastStateDuration ));
            }
        }
    }

    return TRUE;
}

// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{

    // Subscribes to the IRQ service
    IrqHandle = adl_irqSubscribe ( MyTCUHandler, ADL_IRQ_NOTIFY_LOW_LEVEL,
                                   0, ADL_IRQ_OPTION_AUTO_READ );
```

```
    // Subscribes to the TCU service, in Event Detection mode
    TCUHandle = adl_tcuSubscribe ( ADL_TCU_EVENT_DETECTION, IrqHandle, 0,
                                   &Config, NULL );

    // Starts event generation
    adl_tcuStart ( TCUHandle );
}

void MyFunction2 ( void )
{
    // Stops event generation
    adl_tcuStop ( TCUHandle, NULL );

    // Un-subscribes from the TCU service
    adl_tcuUnsubscribe ( TCUHandle );
}
```

3.26.8 The adl_tcuSubscribe Function

This function allows the application to subscribe to the TCU service.

- Prototype

```
s32 adl_tcuSubscribe ( adl_tcuService_e        SrvID,
                       s32                     LowLevelIrqHandle,
                       s32                     HighLevelIrqHandle,
                       void *                  Settings,
                       void *                  Context );
```

- Parameters

SrvID:

Service type to be subscribed, using the **adl_tcuService_e** type.

LowLevelIrqHandle:

Low level interrupt handler identifier, previously returned by the **adl_irqSubscribe** function. This parameter is optional if the HighLevelIrqHandle parameter is supplied..

HighLevelIrqHandle:

High level interrupt handler identifier, previously returned by the **adl_irqSubscribe** function. This parameter is optional if the LowLevelIrqHandle parameter is supplied..

Settings:

TCU service configuration, to be defined according to the SrvID parameter value (Please refer to **adl_tcuService_e** type for more information).

Context:

Pointer on an application context, which will be provided back to the application when the related TCU events will occur.

- Returned values
  - o Handle: A positive TCU service handle on success, to be used in further TCU service function calls.
  - o **ADL_RET_ERR_PARAM** on a supplied parameter error.
  - o **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service was already subscribed for this configuration.
  - o **ADL_RET_ERR_BAD_HDL** if one or both supplied interrupt handler identifiers are invalid.
  - o **ADL_RET_ERR_BAD_STATE** If the function was called in RTE mode (The TCU service is not available in RTE mode).
  - o **ADL_RET_ERR_NOT_SUPPORTED** If the required service is not supported on the current plateform.
  - o **ADL_RET_ERR_SERVICE_LOCKED** If the function was called from a low level interrupt handler (the function is forbidden in this context.

Note:

In some configuration cases, both LowLevelIrqHandle & HighLevelIrqHandle parameters are optional. Please refer to **adl_tcuEventCaptureSettings_t::EventCounter** description for more information.

Whatever is the configuration, events are generated only after a call to the **adl_tcuStart** function.

3.26.9 The adl_tcuUnsubscribe Function

This function allows the application to unsubscribe from the TCU service.

- Prototype

```
s32 adl_tcuUnsubscribe ( s32 Handle );
```

- Parameters

  Handle:

  TCU service handle, previously returned by the **adl_tcuSubscribe** function.

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the supplied TCU handle is unknown.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note:

If the service was started thanks to the **adl_tcuStart** function, an unsubscription operation will implicitely stop it, without having to call the **adl_tcuStop** function.

3.26.10 The adl_tcuStart Function

This function allows the application to start the TCU service event generation. Once started, the related interrupt events are generated, according to the service configuration.
Please refer to the **adl_tcuService_e** type for more information.

- Prototype

```
s32 adl_tcuStart ( s32 Handle );
```

- Parameters

  Handle:

  TCU service handle, previously returned by the **adl_tcuSubscribe** function.

- Returned values
  - **OK** on success.
  - **ADL_RET__ERR_UNKNOWN_HDL** if the supplied TCU handle is unknown.

Note:

If the service was already started, using this function will start it again by reprograming the events generation.

3.26.11 The adl_tcuStop Function

This function allows the application to stop the TCU service event generation. Once stopped, the related interrupt events not are generated anymore. The function has no effect and returns **OK** if the service is already stopped.

- Prototype

```
s32 adl_tcuStop ( s32        Handle
                  void*      OutParam );
```

- Parameters

  Handle:

  TCU service handle, previously returned by the **adl_tcuSubscribe** function.

  OutParam:

  Output parameter of the stop operation, depending on the service type. Please refer to **adl_tcuService_e** type for more information on this parameter usage.

  Whatever is this parameter usage, it is optional and should be set to **NULL**.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied TCU handle is unknown.

3.27 Extint ADL Service

The ADL External Interruption (ExtInt) service allows the application to handle Wireless CPU® External Interruption pin configuration & interruptions.

External interruption pins are multiplexed with the Wireless CPU® GPIO, please refer to the Wireless CPU® Product Technical Specification for more information.

The global External Interruption pin operation is described below:

- The interruption is generated either on:
  - the falling or the rising edge of the input signal, or both.
  - the low or high level of the input signal.
- The input signal is filtered by one of the following processes:
  - Bypass (no filter)
  - Debounce (a stable state is required for a configurable duration before generating the interruption) e.g. EXTINT is the input signal, extint_ch is the generated interruption. When the debounce period equals 4, the Wireless CPU® waits for a stable signal during 4 cycles before generating the interruption.

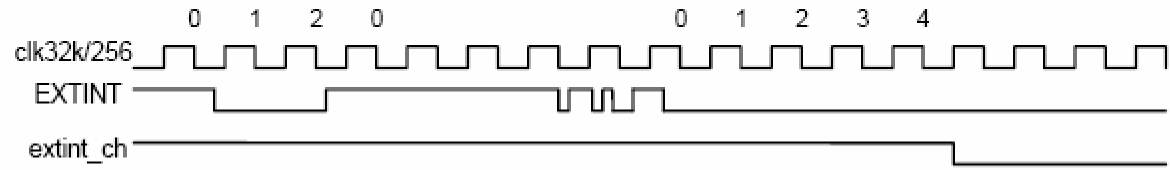


Figure 9: ADL External Interruption Service: Example of Interruption with Debounce Period

  - Stretching (the signal is stretched in order to detect even small glitches in the signal)

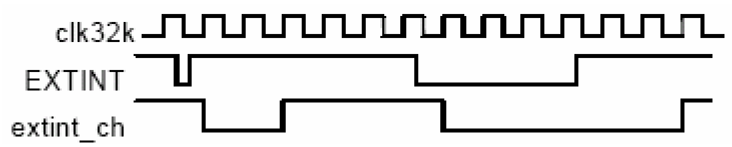


Figure 10: ADL External Interruption Service: Example of Interruption with Stretching Process

e.g. EXTINT is the input signal, extint_ch is the generated interruption. With the stretching process, the generated interruptions are stretched in time, in order not to miss any pulses on the input signal.

- Interruption generated because an External Interruption pin is always pre-acknowledged, whatever is the subscribed option in the IRQ service.

The ADL supplies interface to handle External Interruptions.

The defined operations are:

- A **adl_extintSubscribe** function to subscribe to the External Interruption service.
- A **adl_extintConfig** function to modify an external interruption pin configuration.
- A **adl_extintGetConfig** function to get an external interruption pin configuration.
- A **adl_extintRead** function to retrieve the external interruption pin input status.
- A **adl_extintUnsubscribe** function to unsubscribe from the External Interruption service.

3.27.1 Required Header File

The header file for the ExtInt service definitions is:

```
adl_extint.h
```

3.27.2 The adl_extintConfig_t Structure

This structure allows the application to configure external interruption pin behavior. Using **adl_extintGetCapabilities** to know the available external interruption settings of the Wireless CPU®.

```
typedef struct
{
    adl_extintSensitivity_e  Sensitivity;
    adl_extintFilter_e       Filter;
    u8                       FilterDuration;
    u8                       Pad;            // Internal use only
    void *                   Context
} adl_extintConfig_t;
```

- Fields

Sensitivity:

Interruption generation sensitivity, using the following type:

```
typedef enum
{
 ADL_EXTINT_SENSITIVITY_RISING EDGE,      // Rising edge (edge
                                            sensitivity) interruption
 ADL_EXTINT_SENSITIVITY_FALLING_EDGE,     // Falling edge (edge
                                            sensitivity) interruption
 ADL_EXTINT_SENSITIVITY_BOTH_EDGE,        // Rising & Falling edges (edge
                                          sensitivity)interruption.
                                          ADL_EXTINT_FILTER_STRETCHING_MOD
                                          E cannot be used  with this
                                          mode.
 ADL_EXTINT_SENSITIVITY_LOW LEVEL         // Low level (level sensitivity)
                                          interruption. No Filter can be
                                          used with this mode,
                                          adl_extintConfig_t::Filter value
                                          must be equal to
                                          ADL_EXTINT_FILTER_BYPASS_MODE
 ADL_EXTINT_SENSITIVITY_HIGH LEVEL        // High level(level sensitivity)
                                          interruption. No Filter can be
                                          used with this mode,
                                          adl_extintConfig_t::Filter value
                                          must be equal to
                                          ADL_EXTINT_FILTER_BYPASS_MODE
 ADL_EXTINT_SENSITIVITY_LAST              // Internal use only
} adl_extintSensitivity_e;
```

Filter:

Filter process applied to the input signal:

```
typedef enum
{
 ADL_EXTINT_FILTER_BYPASS_MODE,           // No filter. It is the bypass
                                          mode
 ADL_EXTINT_FILTER_DEBOUNCE_MODE,         // Debounce filter.
                                          adl_extintConfig_t::
                                          FilterDuration value must be
                                          equal to zero.
 ADL_EXTINT_FILTER_STRETCHING_MODE,       // Stretching filter.
                                          adl_extintConfig_t::
                                          FilterDuration value must be
                                          equal to zero.
 ADL_EXTINT_FILTER_LAST                   // Internal use only
} adl_extintFilter_e;
```

FilterDuration:

Time (in number of steps) during which the signal must be stable before generating the interruption. Refers to the function **adl_extintGetCapabilities,** to know the values allowed range.

This parameter is used only with the following filter:

o **ADL_EXTINT_FILTER_DEBOUNCE_MODE**.

Context:

Application context pointer, which will be given back to the application when an interruption event occurs.

3.27.3 The adl_extintInfo_t Structure

This structure allows the application to get the external interrupt pin input status at any time. When an interrupt handler is plugged on the ExtInt service, the SourceData field in the **adl_irqEventData_t** input parameter of this handler must be cast to * **adl_extintInfo_t** type in order to handle the information correctly.

```
typedef struct
{
    u8      PinState;
} adl_extintInfo_t;
```

- Fields

PinState:

External Interruption Pin input status. Current state (0/1) of the input signal plugged on the external interruption pin.

3.27.4 Capabilities

ADL provides informations to get EXTINT capabilities.

The following entries have been defined in the registry:

Registry entry	Type	Description
extint_NbExternalInterrupt	INTEGER	Number of external interrupt pins
extint_RisingEdgeSensitivity	INTEGER	Rising edge sensitivity supported
extint_FallingEdgeSensitivity	INTEGER	Falling edge sensitivity supported
extint_BothEdgeSensitivity	INTEGER	Both edge detector supported
extint_LowLevelSensitivity	INTEGER	Low level sensitivity supported
extint_HighLevelSensitivity	INTEGER	High level sensitivity supported
extint_BypassMode	INTEGER	Bypass mode supported
extint_StretchingMode	INTEGER	Stretching mode supported
extint_DebounceMode	INTEGER	Debounce mode supported
extint_MaxDebounceDuration	INTEGER	Debounce max duration in ms
extint_DebounceNbStep	INTEGER	Number of step for debounce duration
extint_NbPriority	INTEGER	Available priority levels for the EXTINT service (to be used as a **adl_irqPriorityLevel_e** value in the IRQ service)

3.27.5 The adl_extintSubscribe Function

This function allows the application to subscribe to the ExtInt service. Each External Interruption pin can only be subscribed one time. Once subscribed, the pin is no more configurable through the AT commands interface (with AT+WIPC or AT+WFM commands).

Interrupt handlers defined in the IRQ service - using the **adl_irqHandler_f** type - are notified with the following parameters:

- the **Source** parameter will be set to **ADL_IRQ_ID_EXTINT**
- the **adl_irqEventData_t::SourceData** field of the Data parameter has to be casted to an **adl_extintInfo_t** * type, usable to retrieve information about the current external interrupt pin state.
- the **adl_irqEventData_t**::Instance field of the **Data** parameter will have to be considered as an **adl_extintDefsID_t** value, usable to identify which block has raised the current interrupt event.
- the **adl_irqEventData_t::Context** field of the Data parameter will be set to the application context, provided at subscription time.

- Prototype

```
s32 adl_extintSubscribe (     adl_extintID_t           ExtIntID
                              s32                      LowLevelIrqHandle
                              s32                      HighLevelIrqHandle
                              adl_extintConfig_t *     Settings );
```

- Parameters

ExtIntID:

External interruption pin identifier to be subscribed. (see adl_extintID_e).

LowLevelIrqHandle:

Low level interrupt handler identifier, previously returned by the **adl_irqSubscribe** function.

This parameter is optional if the **HighLevelIrqHandle** parameter is supplied.

HighLevelIrqHandle:

High level interrupt handler identifier, previously returned by the **adl_irqSubscribe** function.

This parameter is optional if the **LowLevelIrqHandle** parameter is supplied.

Settings:

External interruption pin configuration, (see section 3.27.2. **adl_extintConfig_t** structure)

- Returned values
  - o A positive or null value on success:
    - **ExtInt** service handle, to be used in further ExtInt service function calls.
  - o A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value
    - **ADL_RET_ERR_NOT_SUPPORTED** if one parameter refers to a mode or a configuration not supported by the Wireless CPU®
    - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service was already subscribed for this external interruption pin (the External Interruption Service can only be subscribed one time for each pin).
    - **ADL_RET_ERR_BAD_HDL** if one or both supplied interrupt handler identifiers are invalid.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

Note:

When interrupt event generated by the EXTINT service are masked (thanks to **adl_irqConfig_t::Enable** field configuration of the IRQ service), events are just delayed until the related handler is enabled again.

3.27.6 The adl_extintConfig Function

This function allows the application to modify an external interruption pin configuration.

- Prototype

```
s32 adl_extintConfig ( s32                    ExtIntHandle,
                       adl_extintConfig_t *   Settings );
```

- Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the **adl_extintSubscribe** function.

Settings:

External interruption pin configuration, (see section 3.27.2. **adl_extintConfig_t** structure).

- Returned values
  - o A **OK** on success.
  - o A negative error value otherwise:
    - ▪ **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - ▪ **ADL_RET_ERR_NOT_SUPPORTED** if one parameter refers to a mode or a configuration not supported by the Wireless CPU®
    - ▪ **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interrupt handle is unknown.

3.27.7 The adl_extintGetConfig Function

This function allows the application to get an external interruption pin configuration.

- Prototype

```
s32 adl_extintGetConfig ( s32                    ExtIntHandle,
                          adl_extintConfig_t *   Settings );
```

- Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the **adl_extintSubscribe** function.

Settings:

External interruption pin configuration, (see section 3.27.2. **adl_extintConfig_t** structure).

- Returned values
  - o A **OK** on success.
  - o A negative error value otherwise:
    - ▪ **ADL_RET_ERR_PARAM** if one parameter has an incorrect value

- **ADL_RET_ERR_UNKNOWN_HDL** if the supplied External Interrupt handle is unknown

3.27.8 The adl_extintRead function

This function allows the application to retrieve the external interruption pin input status.

- Prototype

```
s32 adl_extintRead (s32                    ExtIntHandle,
                    adl_extintInfo_t *     Info );
```

- Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the **adl_extintSubscribe** function.

Info:

External interruption pin information structure (see section 3.27.3 **adl_extintInfo_t** type).

- Returned values
  - A **OK** on success.
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** on a supplied parameter error.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied ExtInt handle is unknown.

3.27.9 The adl_extintUnsubscribe Function

This function allows the application to unsubscribe from the ExtInt service. Associated interrupt handlers are unplugged from the External Interruption source. Pin configuration control is resumed by the AT+WIPC command.

- Prototype

```
s32 adl_extintUnsubscribe (s32     ExtIntHandle );
```

- Parameters

ExtIntHandle:

External Interruption service handle, previously returned by the **adl_extintSubscribe** function.

- Returned values
  - A **OK** on success.
  - A negative error value otherwise:
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.27.10 Example

This example demonstrates how to use the External Interruption service in a nominal case (error cases are not handled).

Complete example using the External Interruption service are also available on the SDK (generic Signal Replica sample).

```
// Global variables

    // use the PIN0 for the Ext Int
    #define EXTINT_PIN0 0

    // ExtInt service handle
    s32 ExtIntHandle;

    // IRQ service handle
    s32 IrqHandle;

    // ExtInt configuration: both edge detection without filter
    adl_extintConfig_t extintConfig =
    { ADL_EXTINT_SENSITIVITY_BOTH_EDGE , ADL_EXTINT_FILTER_BYPASS_MODE ,
      0,0, NULL };


    // ExtInt interrupt handler
    bool MyExtIntHandler ( adl_irqID_e Source, adl_irqNotifyLevel_e
                           NotificationLevel,
                           adl_irqEventData_t * Data )
    {
        // Read the input status
        adl_extintInfo_t Status, * AutoReadStatus;
        adl_extintRead ( ExtIntHandle, &Status );

        // Input status can also be obtained from the auto read option.
        AutoReadStatus = ( adl_extintInfo_t * ) Data->SourceData;

        return TRUE;
    }

    // Somewhere in the application code, used as event handlers
    void MyFunction1 ( void )
    {
        adl_extintCapabilities_t My_ExtInt_Capa;

        adl_extintGetCapabilities ( &My_ExtInt_Capa );

        // Test if the Wireless CPU® have Ext Int pin
        if ( My_ExtInt_Capa.NbExternalInterrupt >= 1 )
        {
            // Subscribes to the IRQ service
            IrqHandle = adl_irqSubscribe ( MyExtIntHandler, ADL IRQ
            NOTIFY_LOW_LEVEL, ADL_IRQ_PRIORITY_HIGH_LEVEL,
            ADL_IRQ_OPTION_AUTO_READ );
```

```
            // Configures comparator channel
            ExtIntHandle = adl_extintSubscribe ( EXTINT_PIN0 , IrqHandle, 0,
&extintConfig );
        }
    }
    void MyFunction2 ( void )
    {
        // Un-subscribes from the ExtInt service
        adl_extintUnsubscribe ( ExtIntHandle );
    }
```

3.28 Execution Context Service

ADL supplies the Execution Context Service interface to handle operations related to the several execution contexts available for an Open AT® application. The application runs under several execution contexts, according to the monitored event (ADL service event, or interrupt event).

The execution contexts are:

- **The application task context;**

  This is the main application context, initialized on the task entry point functions, and scheduled each time a message is received; each message is then converted to an ADL service event, according to its content. This context has a global low priority and should be interrupted by the other ones.

- **The high level interrupt handler context;**

  This is also a task context, but with a higher priority that the main application task. High level interrupt handlers run in this context.

  This context has a global middle priority: when an interrupt raises an event monitored by a high level handler, this context will be immediately activated, even if the application task was running; however, this context could be interrupted by low level interrupt handlers.

- **The low level interrupt handler context;**

  This is a context designed to be activated as soon as possible on an interrupt event.

  This context has a global high priority: when an interrupt raises an event monitored by a low level handler, this context will be immediately activated, even if a task (whatever it is: application task, high level handler or a WAVECOM Firmware task) was running.

  On the other hand, the execution time spent in this context has to be as short as possible; moreover, some service calls are forbidden while this context is running.

As the application code should run in different contexts at the same time, the user should protect his critical functions against re-entrancy. Critical code sections should be protected through a semaphore mechanism (cf. Semaphores service), and/or by temporary disabling interrupts (cf. IRQ service). The ADL services are all re-entrant.

Data can be exchanged between contexts through a message system (cf. Messages service). However, the RAM area is global and accessible from all contexts.

The defined operations of the Execution Context service are:

- Current context identification functions (**adl_ctxGetID** & **adl_ctxGetTaskID**) to retrieve the current context identifiers.
- A Tasks count function (**adl_ctxGetTasksCount**) to retrieve the current tasks count in the runing application.
- A Diagnostic function (**adl_ctxGetDiagnostic**) to retrieve information about the current contexts configuration.
- A State function (**adl_ctxGetState**) to retrieve the required execution context's current state.
- Suspend functions (**adl_ctxSuspend & adl_ctxSuspendExt**) to suspend at any time a running application task.
- Resume functions (**adl_ctxResume & adl_ctxResumeExt**) to resume at any time a suspended application task.
- A Sleep function (**adl_ctxSleep**) to put the current context to sleep for a required duration.

3.28.1 Required Header File

The header file for the Execution Context function is:

```
adl_ctx.h
```

3.28.2 The adl_ctxID_e Type

This type defines the execution context identifiers. Low or High level interrupt handlers, and Wavecom Firmware tasks are identified by specific contants. Application tasks are identified by values between 0 and the **adl_ctxGetTasksCount** function return.

```
typedef enum
{
    ADL_CTX_LOW_LEVEL_IRQ_HANDLER   = 0xFD, //Low level interrupt handler
                                            context
    ADL_CTX_HIGH_LEVEL_IRQ_HANDLER  = 0xFE, // High level interrupt
                                            handler context
    ADL_CTX_ALL                     = 0xFF, // Reserved for internal use
    ADL_CTX_WAVECOM                 = 0xFF, // Wavecom Firmware tasks
                                            context
} adl_ctxID_e;
```

3.28.3 The adl_ctxDiagnostic_e Type

This type defines the available diagnostics, to be retrieved by the **adl_ctxGetDiagnostic** function.

```
typedef enum
{
    ADL_CTX_DIAG_NO_IRQ_PROCESSING              = 0x01,
    ADL_CTX_DIAG_BAD_IRQ_PARAM                  = 0x02,
    ADL_CTX_DIAG_NO_HIGH_LEVEL_IRQ_HANDLER      = 0x04,
} adl_ctxDiagnostic_e;
```

- Description

ADL_CTX_DIAG_NO_IRQ_PROCESSING:	The Open AT IRQ processing mechanism has not been started (interrupt handlers stack sizes have not been supplied).
ADL_CTX_DIAG_BAD_IRQ_PARAM:	Reserved for future use.
ADL_CTX_DIAG_NO_HIGH_LEVEL_IRQ_HANDLER:	High level interrupt handlers are not supported (high level handler stack size is not supplied).

3.28.4 The adl_ctxState_e Type

This type defines the various states for a given execution context, to be retrieved by the **adl_ctxGetState** function.

```
typedef enum
{
    ADL_CTX_STATE_ACTIVE
    ADL_CTX_STATE_WAIT_EVENT
    ADL_CTX_STATE_WAIT_SEMAPHORE
    ADL_CTX_STATE_WAIT_INNER_EVENT
    ADL_CTX_STATE_SLEEPING
    ADL_CTX_STATE_READY
    ADL_CTX_STATE_PREEMPTED
    ADL_CTX_STATE_SUSPENDED
} adl_ctxState_e;
```

- Description

ADL_CTX_STATE_ACTIVE:	The context is currently active (the current code is executed in this context).
ADL_CTX_STATE_WAIT_EVENT:	The context is currently waiting for events (there are currently no events to process).
ADL_CTX_STATE_WAIT_SEMAPHORE:	The context is currently waiting for a semaphore to be produced. The code execution is currently frozen on a

	semaphore consumption function. This can be either an applicative semaphore, or an internal one, consumed within an ADL function call.
ADL_CTX_STATE_WAIT_INNER_EVENT:	The context is currently waiting for an internal event. The code execution is currently frozen, waiting for an internal event within an ADL function call.
ADL_CTX_STATE_SLEEPING:	The context is currently sleeping, after a call to **adl_ctxSleep** function.
ADL_CTX_STATE_READY:	The context has events to process, but is not currently processing them yet, since an higher priority context is processing events.
ADL_CTX_STATE_PREEMPTED:	The context has been pre-empted while it was processing events. It will resume its processing as soon as the higher priority context which is currently running will have terminated his own processing.
ADL_CTX_STATE_SUSPENDED:	The task context is currently suspended, thanks to a call to the adl_ctxSuspend function.

3.28.5 The adl_ctxGetID Function

This function allows the application to retrieve the current execution context identifier.

- Prototype

  adl_ctxID_e adl_ctxGetID (void);

- Returned values
  - o Current application's execution context identifier. Please refer to 3.28.2 **adl_ctxID_e** for more information.
  - o **ID** An application task's zero-based index if the function is called from an ADL service event handler.
  - o **ADL_CTX_LOW_LEVEL_IRQ_HANDLER** if the function is called from a low level interrupt handler.
  - o **ADL_CTX_HIGH_LEVEL_IRQ_HANDLER** if the function is called from a high level interrupt handler.

3.28.6 The adl_ctxGetTaskID Function

This function allows the application to retrieve the current running task identifier:

- In Open AT® task or high level interrupt handler contexts, this function will behave like the **adl_ctxGetID** function.
- In a low level handler execution context, the retrieved identifier will be the active task identifier when the interrupt signal is raised.

- Prototype

```
adl_ctxID_e adl_ctxGetTaskID ( void );
```

- Returned values
  - o Current task's execution context identifier. Please refer to 3.28.2 **adl_ctxID_e** for more information.
  - o **ID** An application task's zero-based index if the function is called from an ADL service event handler.
  - o **ADL_CTX_HIGH_LEVEL_IRQ_HANDLER** if the function is called from a high level interrupt handler.
  - o **Interrupted TaskID** If called from a low level interrupt handler, the returned value depends on the interrupted task:
    - An application task's zero-based index, if an Open AT® application task was running.
    - **ADL_CTX_WAVECOM** if a Wavecom Firmware task was running.
    - **ADL_CTX_HIGH_LEVEL_IRQ_HANDLER** if a high level interrupt handler was running.

3.28.7 The adl_ctxGetTasksCount Function

This function allows the application to retrieve the current application's tasks count.

- Prototype

```
u8 adl_ctxGetTasksCount ( void );
```

- Returned value
  - o Current application's tasks count.

3.28.8 The adl_ctxGetDiagnostic Function

This function allows the application to retrieve information about the current application's execution contexts.

- Prototype

```
u32 adl_ctxGetDiagnostic ( void );
```

- Returned value
  - o Bitwise OR combination of the diagnostics listed in the **adl_ctxDiagnostic_e** type.

3.28.9 The adl_ctxGetState Function

This function allows the application to retrieve the current state of the required execution context.

- Prototype

```
s32  adl_ctxGetState (adl_ctxID_e  Context );
```

- Parameters

 Context:

 Execution context from which the current state has to be queried.

- Returned values
  - o On success, returns the (positive or null) current execution context state, using the **adl_ctxState_e** type.
  - o **ADL_RET_ERR_PARAM** on parameter error.
  - o **ADL_RET_ERR_BAD_HDL** If the low level interrupt handler execution context state is required.

Note:

It is not possible to query the current state of the contexts below (**ADL_RET_ERR_BAD_HDL** error will be returned):

- o the low level interrupt handler execution context (in any case)
- o the high level interrupt handler execution context, if the related **adl_InitIRQHighLevelStackSize** call stack has not be declared in the application.

3.28.10 The adl_ctxSuspend Function

This function allows the application to suspend an application task process. This process can be resumed later thanks to the **adl_ctxResume** function, which should be called from interrupt handlers or from any other application task.

- Prototype

```
s32 adl_ctxSuspend ( adl_ctxID_e Task );
```

- Parameters

 Task:

 Task identifier to be suspended.

 Valid values are in the 0 - **adl_ctxGetTasksCount** range.

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_PARAM** on parameter error.
  - o **ADL_RET_ERR_BAD_STATE** if the required task is already suspended.

Notes:

- o If the function was called in the application task context, it will not return but just suspend the task.

  The **OK** value will be returned when the task process is resumed.
- o While a task is suspended, received events are queued until the process is resumed. If too many events occur, the application mailbox would be overloaded, and this would lead the Wireless CPU® to reset (an application task should not be suspended for a long time, if it is assumed to continue to receive messages).

3.28.11 The adl_ctxSuspendExt Function

This function allows the application to suspend several application tasks processes. Theses process can be resumed later thanks to the **adl_ctxResume** or **adl_ctxResumeExt** functions, which should be called from interrupt handlers or from any other application task.

- Prototype

```
s32 adl_ctxSuspendExt (u32              TasksCount,
                       adl_ctxID_e*     TasksIDArray );
```

- Parameters

  TasksCount:

  Size of the TasksIDArray array parameter (number of tasks to be suspended).

  TasksIDArray:

  Array containing the identifiers of the tasks to be suspended. Valid values are in the 0 - **adl_ctxGetTasksCount** range.

- Returned values
  - o **OK** on success:
  - o **ADL_RET_ERR_PARAM** on parameter error.
  - o **ADL_RET_ERR_BAD_STATE** if the required task is already suspended (no task will be suspended).

Notes:

- o If the function was called in the application task context, it will not return but just suspend the task.

  The **OK** value will be returned when the task process is resumed.
- o While a task is suspended, received events are queued until the process is resumed. If too many events occur, the application mailbox would be overloaded, and this would lead the Wireless CPU® to reset (an application task should not be suspended for a long time, if it is assumed to continue to receive messages).

3.28.12 The adl_ctxResume Function

This function allows the application to resume the Open AT® task process, previously suspended with to the **adl_ctxSuspend** function.

- Prototype

```
s32 adl_ctxResume ( adl_ctxID_e Task );
```

- Parameters

  Task:

  Task identifier to be suspended.

  Valid values are in the 0 - **adl_ctxGetTasksCount** range.

- Returned values
  - **OK** on success:
  - **ADL_RET_ERR_PARAM** on parameter error.
  - **ADL_RET_ERR_BAD_STATE** If the required task is not currently suspended.

Notes:

The required task is resumed as soon as the function is called.

If the resumed task has a lower priority level than the current one, it will be scheduled as soon as the current task process will be over.

If the resumed task has a higher priority level than the current one, it will be scheduled as soon as the function is called.

3.28.13 The adl_ctxResumeExt Function

This function allows the application to resume several Open AT® tasks processes, previously suspended with to the **adl_ctxSuspend** or **adl_ctxSuspendExt** functions.

- Prototype

```
s32 adl_ctxResumeExt (u32                TasksCount,
                      adl_ctxID_e*       TasksIDArray );
```

- Parameters

  TasksCount:

  Size of the TasksIDArray array parameter (number of tasks to be suspended).

  TasksIDArray:

  Array containing the identifiers of the tasks to be suspended. Valid values are in the 0 - **adl_ctxGetTasksCount** range.

- Returned values
  - **OK** on success:
  - **ADL_RET_ERR_PARAM** on parameter error.
  - **ADL_RET_ERR_BAD_STATE** If the required task is not currently suspended (no task will be resumed).

Notes:

The required task is resumed as soon as the function is called.

If the resumed task has a lower priority level than the current one, it will be scheduled as soon as the current task process will be over.

If some resumed task have an higher priority level than the current one, it will be scheduled as soon as the function is called.

3.28.14 The adl_ctxSleep Function

This function allows the application to put the current execution context to sleep for the required duration. This context processing is frozen during this time, allowing other contexts to continue their processing. When the sleep duration expires, the context is resumed and continues its processing.

- Prototype

```
s32 adl_ctxSleep ( u32  Duration );
```

- Parameters

  Duration:

  Required sleep duration, in ticks number (18.5 ms granularity).

- Returned values
  - **OK** on success (when the function returns, the sleep duration has already elapsed).
  - **ADL_RET_ERR_SERVICE_LOCKED** If the function was called from a low level interrupt handler (the function is forbidden in this context).

3.28.15 Example

The code sample below illustrates a nominal use case of the ADL Execution Context Service public interface (error cases are not handled).

```
// Somewhere in the application code, used as an event handler
void MyFunction ( void )
{
    // Get the execution context state
    u32 Diagnose = adl_ctxGetDiagnostic();

    // Get the application tasks count
    u8 TasksCount = adl_ctxGetTasksCount();

    // Get the execution context
    adl_ctxID_e CurCtx = adl_ctxGetID();

    // Check for low level handler context
    if ( CurCtx == ADL_CTX_LOW_LEVEL_IRQ_HANDLER )
    {

        // Get the interrupted context
        adl_ctxID_e InterruptedCtx = adl_ctxGetTaskID();
    }
    else
    {
        // Get the current task state
        adl_ctxState_e State = adl_ctxGetState ( CurCtx );
    }
}


// Somewhere in the application code, used within an high level interrupt
handler
void MyIRQFunction ( void )
{
    // Suspend the first application task
    adl_ctxSuspend ( 0 );

    // Resume the first application task
    adl_ctxResume ( 0 );

    // Put to sleep for some time...
    adl_ctxSleep ( 10 );
}
```

3.29 ADL VariSpeed Service

The ADL VariSpeed service allows the Wireless CPU® clock frequency to be controlled, in order to temporarily increase application performance.

Note:

- The Real Time Enhancement feature must be enabled on the Wireless CPU® in order to make this service available.
- The Real Time Enhancement feature state can be read thanks to the AT+WCFM=5 command response value:
  This feature state is represented by the bit 4 (00000010 in hexadecimal format).
- Please contact your Wavecom distributor for more information on how to enable this feature on the Wireless CPU®.

3.29.1 Required Header File

The header file for the VariSpeed service is:

```
adl_vs.h
```

3.29.2 The adl_vsMode_e Type

This type defines the available CPU modes for the VariSpeed Service.

```
typedef enum
{
  ADL_VS_MODE_STANDARD,
  ADL_VS_MODE_BOOST,
  ADL_VS_MODE_LAST      // Reserved for internal use
} adl_vsMode_e;
```

The **ADL_VS_MODE_STANDARD** constant identifies the standard CPU clock mode (default CPU mode on startup).

The **ADL_VS_MODE_BOOST** constant can be used by the application to make the Wireless CPU® enter a specific boost mode, where the CPU clock frequency is set to its maximum value.

Caution:

In boost mode, the Wireless CPU® power consumption increases significantly. For more information, refer to the Wireless CPU® Power Consumption Mode documentation.

The CPU clock frequencies of the available modes are listed below:

Modes	CPU Clock Frequency
…_STANDARD	26 MHz
…_BOOST	104 MHz"

3.29.3 The adl_vsSubscribe Function

This function allows the application to get control over the VariSpeed service. The VariSpeed service can only be subscribed one time.

- Prototype

```
s32 adl_vsSubscribe ( void );
```

- Parameters

  None

- Returned values
  - A positive or null value on success:
    - VariSpeed service handle, to be used in further service function calls.
  - A negative error value otherwise:
    - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the service has already been subscribed.
    - **ADL_RET_ERR_NOT_SUPPORTED** if the Real Time enhancement feature is not enabled on the Wireless CPU®.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.29.4 The adl_vsSetClockMode Function

This function allows the application to modify the speed of the CPU clock.

- Prototype

```
s32 adl_vsSetClockMode (s32            VsHandle,
                        adl_vsMode_e   ClockMode );
```

- Parameters

  VsHandle:

  VariSpeed service handle, previously returned by the **adl_vsSubscribe** function.

  ClockMode:

  Required clock mode. Refer to **adl_vsMode_e** type definition for more information (see § 3.29.2).

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied handle is unknown.
  - **ADL_RET_ERR_PARAM** if the supplied clock mode value is wrong.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.29.5 The adl_vsUnsubscribe function

This function allows the application to unsubscribe from the VariSpeed service control. The CPU mode is reset to the standard speed.

- Prototype

  s32 adl_vsUnsubscribe (s32 VsHandle);

- Parameters

  VsHandle:

  VariSpeed service handle, previously returned by the **adl_vsSubscribe** function.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_UNKNOWN_HDL** if the supplied handle is unknown.
  - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler (the function is forbidden in this context).

3.29.6 Example

This example demonstrates how to use the VariSpeed service in a nominal case (error cases are not handled).

```
// Global variable: VariSpeed service handle
s32 MyVariSpeedHandle;


// Somewhere in the application code, used as event handlers
void MyFunction1 ( void )
{
    // Subscribe to the VariSpeed service
    MyVariSpeedHandle = adl_vsSubscribe();

    // Enter the boost mode
    adl_vsSetClockMode ( MyVariSpeedHandle, ADL_VS_MODE_BOOST );
}
void MyFunction2 ( void )
{
    // Un-subscribe from the VariSpeed service
    adl_vsUnsubscribe ( MyVariSpeedHandle );
}
```

3.30 ADL DAC Service

The Digital Analog Converter service offers to the customer entities the ability to convert a digital value code of a certain resolution into an analog signal level voltage.

The defined operations are:

- A function **adl_dacSubscribe** to set the reserved DAC parameters.
- A function **adl_dacUnsubscribe** to un-subscribes from a previously allocated DAC handle.
- A function **adl_dacWrite** to allow a DACs to be write from a previously allocated handle.
- A function **adl_dacAnalogWrite** to allow a DAC to be write from a previously allocated handle.
- A function **adl_dacRead** to allow a DAC to be read from a previously allocated handle.
- A function **adl_dacAnalogRead** to allow a DAC to be read from a previously allocated handle.

3.30.1 Required Header File

The header file for the functions dealing with the DAC interface is:

```
adl_dac.h
```

3.30.2 Data Structure

3.30.2.1 The adl_dacParam_t Structure

DAC channel initialization parameters.

- Code

```
typedef struct
{
u32   InitialValue
}adl_dacparam_t
```

- Description

InitialValue

Raw value to set in the register of the DAC.

3.30.3 Defines

3.30.3.1 ADL_DAC_CHANNEL_1

Former constant used to identify the first DAC channel.

```
#define ADL_DAC_CHANNEL_1   0
```

3.30.4 Enumerations

3.30.4.1 The adl_dacType_e

Definition of DAC type.

- Code

```
typedef enum
{
ADL_DAC_TYPE_GEN_PURPOSE  // General Purpose DAC
} adl_dacType_e
```

3.30.5 The adl_dacSubscribe Function

This function subscribes to a DAC channel.

- Prototype

```
s32 adl_dacSubscribe ( u32                Channel,
                       adl_dacParam_t *   DacConfig );
```

- Parameters

  Channel:

  DAC channel identifier.

  DacConfig

  DAC subscription configuration (using **adl_dacParam_t**).

- Returned values
  - o A positive or null value on success:
    - DAC handle to be used on further DAC API functions calls.
  - o A negative error value otherwise (No DAC is reserved):
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the required channel has already been subscribed.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler.
    - **ADL_RET_ERR_NOT_SUPPORTED** if the current Wireless CPU® does not support the DAC service.

3.30.6 The adl_dacUnsubscribe Function

This function un-subscribes from a previously allocated DAC handle.

- Prototype

```
s32 adl_dacUnsubscribe ( s32    DacHandle );
```

- Parameters

DacHandle:

Handle previously returned by **adl_dacSubscribe** function.

- Returned values
  - **OK** on success
  - A negative error value otherwise:
    - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler.

3.30.7 The adl_dacWrite Function

This function writes the digital value on DACs previously allocated.

- Prototype

```
s32 adl_dacWrite       ( s32        DacHandle,
                         u32        DacWrite );
```

- Parameters

DacHandle:

Handle previously returned by **adl_dacSubscribe** function.

DacWrite

New DAC settings to set.

- Returned values
  - **OK** on success
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler and the DAC used cannot be called under interrupt context.

3.30.8 The adl_dacAnalogWrite Function

This function writes a analog value in mV on a DAC previously allocated.

- Prototype

```
s32 adl_dacAnalogWrite ( s32       DacHandle,
                         s32       DacWritemV );
```

- Parameters

DacHandle:

Handle previously returned by **adl_dacSubscribe** function.

DacWritemV

New DAC settings to set (in mV).

- Returned values
  - o **OK** on success
  - o A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler and the DAC used cannot be called under interrupt context.

3.30.9 The adl_dacRead Function

This function reads the last written value on a DAC.

- Prototype

```
s32 adl_dacRead   ( s32       DacHandle,
                    u32*      DacRead );
```

- Parameters

DacHandle:

Handle previously returned by **adl_dacSubscribe** function.

DacRead

DAC digital value.

- Returned values
  - o **OK** on success
  - o A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown

3.30.10 The adl_dacAnalogRead Function

This function reads the last written value on a DAC.

- Prototype

```
s32 adl_dacAnalogRead     ( s32     DacHandle,
                            s32*    DacReadmV );
```

- Parameters

  DacHandle:

  Handle previously returned by **adl_dacSubscribe** function.

  DacReadmV

  DAC analog value in mV.

- Returned values
  - **OK** on success
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the handle is unknown.

3.30.11 Capabilities

ADL provides informations to get DAC capabilities.

The following entries have been defined in the registry:

Registry entry	Type	Description
dac_NbBlocks	INTEGER	The number of DAC blocks available
dac_xx_DigitInitValue	INTEGER	Digital value at DAC resource allocation. dac_xx_DigitInitValue is set at -1 if the default value is unknown.
dac_xx_MaxRefVoltage	INTEGER	Reference voltage of the DAC output when the maximal digital value is set.
dac_xx_MinRefVoltage	INTEGER	Reference voltage of the DAC output when the minimal digital value is set.
dac_xx_Resolution	INTEGER	DAC resolution in steps.
dac_xx_DacType	INTEGER	DAC type, see 3.30.4.1 **adl_dacType_e**.
dac_xx_InterruptContextUsed	INTEGER	This value is set to 1 if DAC write operations can be called under interrupt context

Notes:

- For the registry entry the **xx** part must be replaced by the number of the instance.
  Example: if you want the Resolution capabilities of the DAC02 block, the registry entry to use will be **dac_02_Resolution**.
- DACs will be identified with a number as 0, 1, 2, ... dac_NbBlocks-1.
- For each block, the settling time capabilities are defined in the PTS.

3.30.12 Example

The sample DAC illustrates a nominal use case of the ADL DAC Service public interface.

```
// Global variable
    s32 MyDACHandle;
    u32 MyDACID = 1;

    …

    // Somewhere in the application code, used as an event handler
    void MyFunction ( void )
    {
        // Initialization structure
        adl_dacParam_t InitStruct = { 0 };

        // Subscribe to the DAC service
        MyDACHandle = adl_dacSubscribe ( MyDACID , &InitStruct );

        // Write a value on the DAC block
        adl_dacWrite ( MyDACHandle, 80 );

        ...

        // Write another value on the DAC block
        adl_dacWrite ( MyDACHandle, 190 );

        ...

        // Write a analog value on the DAC block (1500 mV)
        adl_dacAnalogWrite ( MyDACHandle, 1500 );

        ...

        {
            s32 AnalogValue;
            // Read the last analog value write on the DAC block
            adl_dacAnalogRead ( MyDACHandle , &AnalogValue );

            ...
     }
```

```
         ...

        {
            u32 Value;
            // Read the last register value write on the DAC block
            adl_dacRead ( MyDACHandle , &Value );

            ...
         }

        // Unsubscribe from the DAC service
        adl_dacUnsubscribe ( MyDACHandle );
    }
```

3.31 ADL ADC Service

The goal of the ADC service is to offer all the interfaces to handle application using ADC for voltage level measurement such as temperature and battery level monitoring purposes. The ADC interface provides also a way to get analog value from various sources. The ADC is a circuit section that converts low frequency analog signals, like battery voltage or temperature, to digital value.

The defined operations are:

- A function **adl_adcRead** to read a ADC register value.
- A function **adl_adcAnalogRead** to read a ADC analog value in mV.

3.31.1 Required Header File

The header file for the functions dealing with the ADC interface is:

```
adl_adc.h
```

3.31.2 The adl_adcRead Function

This function allows ADCs to be read. For this operation, it is not necessary to subscribe to ADC previously.

- Prototype

```
s32 adl_adcRead  (u32               ChannelID,
                  u32*              AdcRawValue );
```

- Parameters

  ChannelID:

  Channel ID of the ADC to read.

  AdcRawValue

  The value of the ADC register.

- Returned values
  - A **OK** on success (read values are updated in the **AdcRawValue** parameter)
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler and the ADC used cannot be called under interrupt context.

3.31.3 The adl_adcAnalogRead Function

This function allows ADCs to be read. For this operation, it is not necessary to subscribe to ADC previously.

- Prototype

```
s32 adl_adcAnalogRead    (u32      ChannelID,
                          s32*     AdcValuemV );
```

- Parameters

  ChannelID:

  Channel ID of the ADC to read.

  AdcValuemV

  The value corresponding to the register Value of the ADC voltage in mV.

- Returned values
  - A **OK** on success (read values are updated in the **AdcValuemV** parameter)
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if one parameter has an incorrect value.
    - **ADL_RET_ERR_SERVICE_LOCKED** if the function was called from a low level interrupt handler and the ADC used can not be called under interrupt context.

3.31.4 Capabilities

ADL provides informations to get ADC capabilities.

The following entries have been defined in the registry:

Registry entry	Type	Description
adc_NbBlocks	INTEGER	The number of ADC blocks available
adc_xx_ResolutionsBits	INTEGER	To get on how many bits, is coded the result.
adc_xx_ MaxInputRange	INTEGER	The minimum input voltage in mV supported by each ADC.
adc_xx_ MinInputRange	INTEGER	The maximum input voltage in mV supported by each ADC.
adc_xx_InterruptContextUsed	INTEGER	This value is set to 1, if ADC read functions can be called under interrupt context

Notes:

- For the registry entry the **xx** part must be replaced by the number of the instance.
  Example: if you want the Resolution Bits capabilities of the ADC02 block the registry entry to use will be **adc_02_ResolutionBits**.
- ADCs will be identified with a number as 0, 1, 2, ... adc_NbBlocks-1.
- For each block, the sampling time capability is defined in the PTS.

3.31.5 Example

The code sample below illustrates a nominal use case of the ADL ADC Service public interface (error cases are not handled).

```
// ADC read functions

    // Read ADC Raw Value
    u32 My_adcReadRawValue ( u32 My_adcID )
    {
        // Variable to store ADC voltage information
        u32 My_adcValue;

        // Read the ADC
        adl_adcRead ( My_adcID , &My_adcValue );

        return ( My_adcValue );
    }

    // Read ADC value in mV
    u32 My_adcReadValue ( u32 My_adcID )
    {
        // Variable to store ADC voltage information
        s32 My_adcValue_mV;

        // Read the ADC
        adl_adcAnalogRead ( My_adcID , &My_adcValue_mV );

        return ( My_adcValue_mV );
    }
```

3.32 ADL Queue Service

ADL supplies this interface to provide to applications thread-safe queue service facilities, usable from any execution context.

The defined operations are:

- A subscription function **adl_queueSubscribe** to create a queue resource.
- An unsubscription function **adl_queueUnsubscribe** to delete a queue resource.
- A state query function **adl_queueIsEmpty** to check if it remains items in the queue.
- item handling functions **adl_queuePushItem** & **adl_queuePopItem** to queue and de-queue items.

3.32.1 Required Header File

The header file for the functions dealing with the Queue interface is:

```
adl_queue.h
```

3.32.2 The adl_queueOptions_e Type

This type allows to define the behaviour of a queue resource.

```
typedef enum
{
    ADL_QUEUE_OPT_FIFO,
    ADL_QUEUE_OPT_LIFO,
    ADL_QUEUE_OPT_LAST          //Reserved for internal use
} adl_queueOptions_e;
```

- Description

ADL_QUEUE_OPT_FIFO:	First In, First Out: the first pushed item will be retrieved first.
ADL_QUEUE_OPT_LIFO:	Last In, First Out: the last pushed item will be retrieved first.

3.32.3 The adl_queueSubscribe Function

This function allows the application to create a thread-safe queue resource. The obtained handle is then usable with the other service operations.

- Prototype

```
s32 adl_queueSubscribe  (adl_queueOptions_e    Option);
```

- Parameter

  Option

  Allows to configure the behaviour of the queue resource, using one of the **adl_queueOptions_e** type values.

- Returned values
  - **Handle** A positive queue service handle on success.
  - **ADL_RET_ERR_PARAM** on parameter error.
  - **ADL_RET_ERR_SERVICE_LOCKED** If the function was called from a low level interrupt handler (the function is forbidden in this context).

3.32.4 The adl_queueUnsubscribe Function

This function allows the application to release a previously subscribed queue resource, if this one is empty.

- Prototype

```
s32 adl_queueUnsubscribe  (s32     Handle );
```

- Parameters

  Handle:

  A queue service handle, previously returned by the **adl_queueSubscribe** function.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_BAD_STATE** If the provided queue resource is not empty; it shall be firstly emptied thanks to the **adl_queuePopItem** function.
  - **ADL_RET_ERR_SERVICE_LOCKED** If the function was called from a low level interrupt handler (the function is forbidden in this context).

3.32.5 The adl_ queueIsEmpty Function

This function informs the application, if items remain in the provided queue.

- Prototype

```
s32 adl_queueIsEmpty  (s32   Handle );
```

- Parameters

Handle:

A queue service handle, previously returned by the **adl_queueSubscribe** function.

- Returned values
  - o **FALSE** If it remains at least one item in the queue
  - o **TRUE** If the queue is empty.
  - o **ADL_RET_ERR_UNKNOWN_HANDLE** If the provided handle is invalid.

3.32.6 The adl_ queuePushItem Function

This function allows the application to add an item at the end of the provided queue resource.

- Prototype

```
s32 adl_queuePushItem  (s32       Handle,
                         void*     Item );
```

- Parameters

Handle:

A queue service handle, previously returned by the **adl_queueSubscribe** function.

Item

Pointer on the application item; this parameter cannot be **NULL**

- Returned values
  - o **OK** on success.
  - o **ADL_RET_ERR_UNKNOWN_HANDLE** If the provided handle is invalid.
  - o **ADL_RET_ERR_PARAM** on parameter error (Bad item pointer).
- Exceptions
  - o **144:** Raised if too many items are pushed in the queue.

Note:

This function is thread-safe, and shall be called from any execution context.

This means that operations on queue items are performed under a critical section, in which the current context cannot be pre-empted by any other context.

3.32.7 The adl_ queuePopItem Function

This function allows the application to retrieve an item from the provided queue resource, according to the defined behaviour at subscription time (cf. **adl_queueSubscribe** function):

- If the queue option is **ADL_QUEUE_OPT_FIFO**, the first pushed item is retrieved by the function
- If the queue option is **ADL_QUEUE_OPT_LIFO**, the last pushed item is retrieved by the function.

- Prototype

```
void* adl_queuePopItem  (s32       Handle );
```

- Parameters

Handle:

A queue service handle, previously returned by the **adl_queueSubscribe** function.

- Returned values
  - **Item** on success, a pointer on the de-queued item.
  - **NULL** If the provided handle is unknown, or if the related queue is empty.

Note:

This function is thread-safe, and shall be called from any execution context.

This means that operations on queue items are performed under a critical section, in which the current context cannot be pre-empted by any other context.

3.32.8 Example

The code sample below illustrates a nominal use case of the ADL ADC Service public interface (error cases are not handled).

```
// Event handler, somewhere in the application
void MyFunction ( void )
{
    // Queue handle
    s32 MyHandle;

    // Queue state
    s32 State;

    // Item definitions
    u32 MyItem1, MyItem2, *GotItem1, *GotItem2;

    // Create a FIFO queue resource
    MyHandle = adl_queueSubscribe(ADL_QUEUE_OPT_FIFO);

    // Check the queue state (shall be empty)
    State = adl_queueIsEmpty ( MyHandle );

    // Push items
    adl_queuePushItem ( MyHandle, &MyItem1 );
    adl_queuePushItem ( MyHandle, &MyItem2 );

    // Check the queue state (shall not be empty)
    State = adl_queueIsEmpty ( MyHandle );

    // Pop items (retrieved in FIFO order)
    GotItem1 = adl_queuePopItem ( MyHandle );
    GotItem2 = adl_queuePopItem ( MyHandle );

    // Check the queue state (shall be empty)
    State = adl_queueIsEmpty ( MyHandle );

    // Delete the queue resource
    adl_queueUnsubscribe ( MyHandle );
}
```

3.33 ADL Audio Service

The ADL Audio Service allows to handle audio resources, and play or listen supported audio formats on these resources (single/dual tones, DTMF tones, melodies, PCM audio streams, decoded DTMF streams).

The defined operations are:

- An **adl_audioSubscribe** function to subscribe to an audio resource.
- An **adl_audioUnsubscribe** function to unsubscribe from an audio resource.
- An **adl_audioTonePlay** function to play a single/dual tone.
- An **adl_audioDTMFPlay** function to play a DTMF tone.
- An **adl_audioMelodyPlay** function to play a melody.
- An **adl_audioTonePlayExt** function to play a single/dual tone (extension).
- An **adl_audioDTMFPlayExt** function to play a DTMF tone (extension).
- An **adl_audioMelodyPlayExt** function to play a melody (extension).
- An **adl_audioStreamPlay** function to play an audio stream.
- An **adl_audioStreamListen** function to listen to an audio stream.
- An **adl_audioStop** function to stop playing or listening.
- An **adl_audioSetOption** function to set audio options.
- An **adl_audioGetOption** function to get audio options

3.33.1 Required Header File

The header file for the functions dealing with the Audio service interface is:

adl_audio.h

3.33.2 Data Structures

3.33.2.1 The adl_audioDecodedDtmf_u Union

This union defines different types of buffers which are used according to the decoding mode (Raw mode enable or disable) when listening to an audio DTMF stream.
(refer to 3.33.4.5 **ADL_AUDIO_DTMF_DETECT_BLANK_DURATION** for more information about Raw mode).

- Code

```
typedef union
{
  ascii                                DecodedDTMFChars
                                       [ADL_AUDIO_MAX_DTMF_PER_FRAME]
 adl_audioPostProcessedDecoder_t       PostProcessedDTMF
} adl_audioDecodedDtmf_u;
```

- Description

DecodedDTMFChars:

This field contains decoded DTMF in Raw mode.

PostProcessedDTMF:

This field contains informations about decoded DTMF and decoding post-process. (Refer to **adl_audioPostProcessedDecoder_t** for more information).

3.33.2.2 The adl_audioPostProcessedDecoder_t Structure

This structure allows the application to handle post-processed DTMF datas when listening to an audio DTMF stream with Raw mode deactivated. (Refer to 3.33.4.5 **ADL_AUDIO_DTMF_DETECT_BLANK_DURATION** for more information about Raw mode).

- Code

```
typedef struct
{
  u32       Metrics;
  u32       Duration;
  ascii     DecodedDTMF
} adl_audioPostProcessedDecoder_t;
```

- Description

Metrics:

Processing metrics, contains informations about DTMF decoding process. Reserved for Future Use.

Duration:

DTMF duration, contains post-processed DTMF duration, in ms

DecodedDTMF:

PostProcessed DTMF buffer contains decoded DTMF.

3.33.2.3 The adl_audioStream_t Structure

This structure allows the application to handle data buffer according to the audio format when an audio stream interrupt occurs during a playing (**adl_audioStreamPlay**) or a listening to (**adl_audioStreamListen**) an audio stream.

- Code

```
typedef struct
{
  adl_audioFormats_e              audioFormat
  adl_audioStreamDataBuffer_u *   DataBuffer
  bool *                          BufferReady
  bool *                          BufferEmpty
} adl_audiostream_t;
```

- Description

audioFormat:

Stream audio format (refer to **adl_audioFormats_e** for more information)

DataBuffer:

Audio data exchange buffer:

- This field stores audio sample during an audio PCM stream listening or decoded DTMF during an audio DTMF stream listening.
- It contains audio sample to play during an audio PCM stream playing. (Refer to 3.33.2.4 **adl_audioStreamDataBuffer_u** structure for more information).

BufferReady:

When an audio stream is played, each time an interrupt occurs this flag has to set to TRUE when data buffer is filled. If this flag is not set to TRUE, an 'empty' frame composed of 0x0 will be sent and set the BufferEmpty flag to TRUE. Once the sample is played, BufferReady is set to FALSE by the firmware. **This pointer is initialized only when an audio stream is played. Currently, it is used only for PCM stream playing.**

BufferEmpty:

When an audio stream is played, this flag is set to TRUE when empty data buffer is played (for example, when an interrupt is missing). This flag is used only for information and it has to be set to FALSE by application. **This pointer is initialized only when an audio stream is played. Currently, it is used only for PCM stream playing.**

3.33.2.4 The adl_audioStreamDataBuffer_u Union

This union defines different types of buffers, which are used according to the audio format when an audio stream interrupt occurs.

- Code

```
typedef union
{
  u8                              PCMData [1]
  adl_audioDecodedDtmf_u          DTMFData
} adl_audiostreamDataBuffer_u;
```

- Description

PCMData [1]:

PCM stream data buffer.

This buffer is used when playing or listening to an audio PCM stream.

DTMFData:

DTMF stream data buffer.

This buffer stores decoded DTMF when listening to an audio DTMF stream according to the decoding mode which is used. Please refer to 3.33.2.1 **adl_audioDecodedDtmf_u** for more information about DTMF buffer structure and 3.33.4.5 **ADL_AUDIO_DTMF_DETECT_BLANK_DURATION** for more information about decoding modes.

3.33.3 Defines

3.33.3.1 ADL_ AUDIO_MAX_DTMF_PER_FRAME

This constant defines maximal number of received DTMFs each time interrupt handlers are called when a listening to a DTFM stream in Raw mode (Refer to 3.33.4.5 **ADL_AUDIO_DTMF_DETECT_BLANK_DURATION** for more information about Raw mode).

- Code:

```
#define ADL_AUDIO_MAX_DTMF_PER_FRAME   2
```

3.33.3.2 ADL_ AUDIO_NOTE_DEF

This macro is used to define the note value to play according to the note definition, the scale and the note duration.

To play a melody, each note defines in the melody buffer has to be defined with this macro (see 3.33.7.3 **adl_audioMelodyPlay** function).

- Code:

```
#define ADL_AUDIO_NOTE_DEF (ID,
                            Scale,
                            Duration )(((ID)+(Scale*12))<<8)+(Duration));
```

- Parameters

ID :

This parameter corresponds to the note identification. Please refer to the code below for the Group Notes identification for melody.

```
#define ADL_AUDIO_C   0x01        //C
#define ADL_AUDIO_CS  0x02        //C #
#define ADL_AUDIO_D   0x03        //D
#define ADL_AUDIO_DS  0x04        //D #
#define ADL_AUDIO_E   0x05        //E
#define ADL_AUDIO_F   0x06        //F
#define ADL_AUDIO_FS  0x07        //F #
#define ADL_AUDIO_G   0x08        //G
#define ADL_AUDIO_GS  0x09        //G #
#define ADL_AUDIO_A   0x0A        //A
#define ADL_AUDIO_AS  0x0B        //A #
define ADL_AUDIO_B    0x0C        //B
#define DL_AUDIO_NO_SOUND 0xFF    //No sound
```

Scale:

This parameter defines the note scale (0 - 7).

Duration:

This parameter defines the note duration. Please refer to the Group Notes Durations code below to see the set of note durations which are available.

```
#define ADL_AUDIO_WHOLE_NOTE           0x10  //Whole note
#define ADL_AUDIO_HALF                 0x08  //Half note
#define ADL_AUDIO_QUARTER              0x04  //Quarter note
#define ADL_AUDIO_HEIGHT               0x02  //Height note
#define ADL_AUDIO_SIXTEENTH            0x01  //Sixteenth note
#define ADL_AUDIO_DOTTED_HALF          0x0C  //Dotted half note
#define ADL_AUDIO_DOTTED_QUARTER       0x06  //Dotted quarter
#define ADL_AUDIO_DOTTED_HEIGHT        0x03  //Dotted height
```

3.33.4 Enumerations

3.33.4.1 The adl_ audioResources_e Type

This type lists the available audio resources of the Wireless CPU®, including the local ones (plugged to the Wireless CPU® itself) and the ones related to any running voice call. These resources are usable either to play a pre-defined/stream audio format (output resources), or to listen to an incoming audio stream (input resources).

- Code

```
typedef enum
{
            ADL_AUDIO_SPEAKER,
            ADL_AUDIO_BUZZER,
            ADL_AUDIO_MICROPHONE,
            ADL_AUDIO_VOICE_CALL_RX,
            ADL_AUDIO_VOICE_CALL_TX
} adl_audioResources_e;
```

- Description

ADL_AUDIO_SPEAKER:	Current speaker (output resource; please refer to the AT Command interface guide for more information on how to select the current speaker).
ADL_AUDIO_BUZZER:	Buzzer (output resource, just usable to play single frequency tones & melodies).
ADL_AUDIO_MICROPHONE:	Current microphone (input resource; please refer to the AT Command interface guide for more information on how to select the current microphone).
ADL_AUDIO_VOICE_CALL_RX:	Running voice call incoming channel (input resource, available when a voice call is running to listen to audio streams).
ADL_AUDIO_VOICE_CALL_TX:	Running voice call outgoing channel (output resource, available when a voice call is running to play audio streams).

3.33.4.2 The adl_audioResourceOption_e Type

This type defines the audio resource subscription options.

- Code

```
typedef enum
{
            ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION = 0x00,
            ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION  = 0x01
} adl_audioResourceOption_e;
```

- Description

ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION:

Never allows prioritary uses of the resource (the resource subscriber owns the resource until unsubscription time).

ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION:

Allows prioritary uses of the resource (such as incoming voice call melody, outgoing voice call tone play, SIM Toolkit application tone play).

3.33.4.3 The adl_audioFormats_e Type

This type defines the audio stream formats for audio stream playing/listening processes.

- Code

```
typedef enum
{
   ADL_AUDIO_DTMF                     //Decoded DTMF sequence
   ADL_AUDIO_PCM_MONO_8K_16B
} adl_audioFormats_e;
```

ADL_AUDIO_PCM_MONO_8K_16B:

PCM mono 16 bits / 8 KHz Audio sample.

3.33.4.4 The adl_audioEvents_e Type

Set of events that will be notified by ADL to audio event handlers.

- Code

```
typedef enum
{
  ADL_AUDIO_EVENT_NORMAL_STOP,
  ADL_AUDIO_EVENT_RESOURCE_RELEASED
} adl_audioEvents_e;
```

- Description

ADL_AUDIO_EVENT_NORMAL_STOP:

A pre-defined audio format play has ended (please refer to 3.33.7.3 **adl_audioDTMFPlay**, **adl_audioTonePlay** or **adl_audioMelodyPlay** for more information). This event is not sent on a request to stop from application.

ADL_AUDIO_EVENT_RESOURCE_RELEASED:

Resource has been automatically unsubscribed due to a prioritary use by the Wireless CPU® (please refer to the **ADL_AUDIO_RESOURCE_OPTION_ALLOW_PREEMPTION** option and **adl_audioSubscribe** for more information).

3.33.4.5 The adl_audioOptionTypes_e Type

This type includes a set of options readable and writable through the **adl_audioSetOption** and **adl_audioGetOption** functions. These options allow to configure the Wireless CPU® audio service behaviour, and to get this audio service capabilities and parameters ranges.

For each option, the value type is specified, and a specific keyword indicates the option access:

- R: the option is only readable.
- RW: the option is both readable & writable.

Note:

For more information about indicative values which should be returned when reading options for MIN/MAX values, please refer to the Audio Commands chapter of the AT Commands Interface Guide

- Code

```
typedef enum
{
  ADL_AUDIO_DTMF_DETECT_BLANK_DURATION,
  ADL_AUDIO_MAX_FREQUENCY,
  ADL_AUDIO_MIN_FREQUENCY,
  ADL_AUDIO_MAX_GAIN,
  ADL_AUDIO_MIN_GAIN,
  ADL_AUDIO_MAX_DURATION,
  ADL_AUDIO_MIN_DURATION,
  ADL_AUDIO_MAX_NOTE_VALUE,
  ADL_AUDIO_MIN_NOTE_VALUE,
  ADL_AUDIO_DTMF_STREAM_BUFFER_SIZE,
  ADL_AUDIO_DTMF_PROCESSED_STREAM_BUFFER_SIZE,
  ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE
} adl_audioOptionTypes_e;
```

* Description

ADL_AUDIO_DTMF_DETECT_BLANK_DURATION

RW: DTMF decoding option (u16); it allows to define the blank duration (ms) in order to detect the end of a DTMF. This value will act on the Wireless CPU® behaviour to return information about DTMF when listening to a DTMF audio stream. The value has to be a 10-ms multiple. If a NULL value is specified, DTMF decoder will be in Raw mode (default), Raw datas coming from DTMF decoder are sent every 20 ms via interrupt handlers. This mode implies to implement an algorithm in order to detect the good DTMF. (Refer to 3.33.2.2 **adl_audioDecodedDtmf_u** for more information about buffer type used). Otherwise the Raw mode is disabled. The value specifies the blank duration which notifies the end of DTMF. Each time a DTMF is detected, interrupt handlers are called. Please refer to **adl_audioPostProcessedDecoder_t** structure for more information about datas stored.

ADL_AUDIO_MAX_FREQUENCY

R: allows to get the maximum frequency allowed to be played on the required output resource (please refer to **adl_audioResourceOption_e** for more information, section 3.33.4.2). The returned frequency value is defined in Hz (u16).

ADL_AUDIO_MIN_FREQUENCY

R: allows to get the minimum frequency allowed to be played on the required output resource (please refer to **adl_audioResourceOption_e** for more information, section 3.33.4.2). The returned frequency value is defined in Hz (u16).

ADL_AUDIO_MAX_GAIN

R: supplies the maximum gain which can be set to play a pre-defined audio format (please refer to **adl_audioDTMFPlayExt**, **adl_audioTonePlayExt** or **adl_audioMelodyPlayExt** for more information, section 3.33.7.3). The returned gain value is defined in 1/100 of dB (s16).

ADL_AUDIO_MIN_GAIN

R: supplies the minimum gain which can be set to play a pre-defined audio format (please refer to **adl_audioDTMFPlayExt**, **adl_audioTonePlayExt** or **adl_audioMelodyPlayExt** for more information, section 3.33.7.3). The returned gain value is defined in 1/100 of dB (s16).

ADL_AUDIO_MAX_DURATION

R: supplies the maximum duration which can be set to play a DTMF tone or a single/dual tone (please refer to **adl_audioDTMFPlay** or **adl_audioTonePlay** for more information, section 3.33.7.3). The returned duration value is defined in ms (u32).

ADL_AUDIO_MIN_DURATION

R: supplies the minimum duration which can be set to play a DTMF tone or a single/dual tone (please refer to **adl_audioDTMFPlay** or **adl_audioTonePlay** for more information, section 3.33.7.3). The returned duration value is defined in ms (u32).

ADL_AUDIO_MAX_NOTE_VALUE

R: supplies the maximum duration for a note (tempo) which can be set to play play a melody (please refer to **adl_audioMelodyPlay** for more information, section 3.33.7.3). This value is the maximal value which can be defined with **ADL_AUDIO_NOTE_DEF** macro (u32).

ADL_AUDIO_MIN_NOTE_VALUE

R: supplies the minimum duration for a note (tempo) which can be set to play play a melody (please refer to **adl_audioMelodyPlay** for more information, section 3.33.7.3). This value is the minimal value which can be defined with **ADL_AUDIO_NOTE_DEF** macro (u32).

ADL_AUDIO_DTMF_RAW_STREAM_BUFFER_SIZE

R: allows to get the buffer type to allocate for listening to a DTMF stream in Raw mode or playing a DTMF stream, defined in number of bytes (u8).

ADL_AUDIO_DTMF_PROCESSED_STREAM_BUFFER_SIZE

R: allows to get the buffer type to allocate for listening to a DTMF stream in Pre-processed mode, defined in number of bytes (u8).

ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE

R: allows the user to get the buffer type to allocated for playing or listening to on a PCM 8KHz 16 bits Mono stream, defined in number of bytes (u8).

3.33.5 Audio events handler

This call-back function has to be supplied to ADL through the **adl_audioSubscribe** interface in order to receive audio resource related events

- prototype

```
typedef void(*) adl_audioEventHandler_f(s32                  audioHandle,
                                        adl_audioEvents_e    Event);
```

- parameters

audioHandle

This is the handle of the audio resource which is associated to the event (refer to **adl_audioSubscribe** for more information about the audio resource handle, section 3.33.6.1).

Event

This is the received event identifier (refer to **adl_audioEvents_e** for more information about the different events, section 3.33.4.4).

3.33.6 Audio resources control

3.33.6.1 The adl_audioSubscribe Function

This function allows to subscribe to the one of the available resources and specify its behaviour when another client attempts to subscribe it.
A call-back function is associated for audio resources related events, the **adl_audioPostProcessedDecoder_t** Type.

- Prototype

```
s32 adl_audioSubscribe (adl_audioResources_e        audioResource,
                        adl_audioEventHandler_f     audioEventHandler,
                        adl_audioResourceOption_e   Options );
```

- Parameters

  audioResource

  Requested audio resource.

  audioEventHandler

  Application provided audio event call-back function (refer to **adl_audioEventHandler_f** for more information.

  Options

  Option about the audio resource behaviour (refer to 3.33.4.2 **adl_audioResourceOption_e** for more information).

- Returned values
  - Positive or NULL if allocation succeeds, to be used on further audio API functions calls.
  - **ADL_RET_ERR_PARAM** if the parameter has an incorrect value.
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the resource is already subscribed.
  - **ADL_RET_ERR_NOT_SUPPORTED** if the resource is not supported.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note:

ERROR values are defined in **adl_error.h**.

3.33.6.2 The adl_audioUnsubscribe Function

This function allows to unsubscribe to one of the resources which have been previously subscribed.

A resource cannot be unsubscribed if it is running, process on this resource has to be previously stopped (refer to **adl_audioStop** for more information, section 3.33.9.1).

- Prototype

```
s32 adl_audioUnsubscribe (s32 audioHandle );
```

- Parameter

  audioHandle

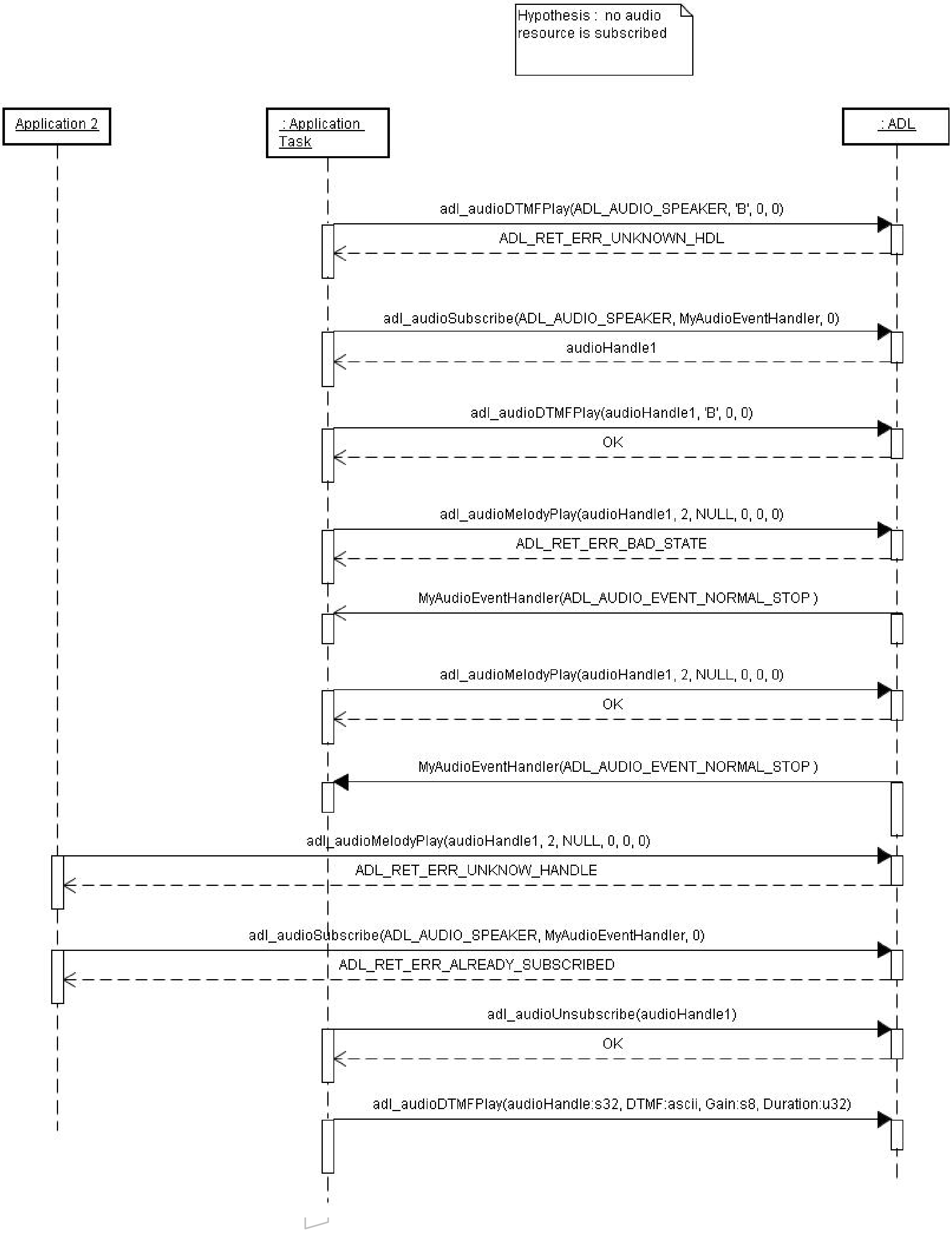
  Handle of the audio resource which has to be unsubscribed.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - **ADL_RET_ERR_NOT_SUBSCRIBED** if no audio resource has been subscribed.
  - **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.33.7 Play a pre-defined audio format

These functions allow to play a melody, a tone or a DTMF on the available audio outputs.

The following diagram illustrates a typical use of the ADL Audio Service interface to play a predefined audio format.


Hypothesis : no audio resource is subscribed
Application 2
: Application Task
: ADL
adl_audioDTMFPlay(ADL_AUDIO_SPEAKER, 'B', 0, 0)
ADL_RET_ERR_UNKNOWN_HDL
adl_audioSubscribe(ADL_AUDIO_SPEAKER, MyAudioEventHandler, 0)
audioHandle1
adl_audioDTMFPlay(audioHandle1, 'B', 0, 0)
OK
adl_audioMelodyPlay(audioHandle1, 2, NULL, 0, 0, 0)
ADL_RET_ERR_BAD_STATE
MyAudioEventHandler(ADL_AUDIO_EVENT_NORMAL_STOP)
adl_audioMelodyPlay(audioHandle1, 2, NULL, 0, 0, 0)
OK
MyAudioEventHandler(ADL_AUDIO_EVENT_NORMAL_STOP)
adl_audioMelodyPlay(audioHandle1, 2, NULL, 0, 0, 0)
ADL_RET_ERR_UNKNOW_HANDLE
adl_audioSubscribe(ADL_AUDIO_SPEAKER, MyAudioEventHandler, 0)
ADL_RET_ERR_ALREADY_SUBSCRIBED
adl_audioUnsubscribe(audioHandle1)
OK
adl_audioDTMFPlay(audioHandle:s32, DTMF:ascii, Gain:s8, Duration:u32)

3.33.7.1 The adl_audioTonePlay Function

This function plays a single or dual tone on current speaker and only a single tone on buzzer.
Only the speaker output is able to play tones in two frequencies. The second tone parameters are ignored on buzzer output.
The specified output stops to play at the end of tone duration or on an application request (refer to **adl_audioStop** for more information section 3.33.9.1).
Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT commands Interface User Guide [1] for more information.

- Prototype

```
s32 adl_audioTonePlay (s32          audioHandle,
                       u16          Frequency1,
                       s8           Gain1,
                       u16          Frequency2,
                       s8           Gain2,
                       u32          Duration );
```

- Parameters

audioHandle

Handle of the audio resource which will play tone (current speaker or buzzer).

Frequency1

Frequency for the 1st tone (Hz).

Gain1

This parameter sets the tone gain which will be applied to the 1st frequency value (dB).

Frequency2

Frequency for the 2nd tone (Hz), only processed on current speaker. Frequency2 has to set to 0 to play a single tone on current speaker.

Gain2

This parameter sets the tone gain which will be applied to the 2nd frequency value (dB).

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple.

- Returned values
    - **OK** on success.
    - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
    - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
    - **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.

- o **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for tone playing.
- o **ADL_RET_ERR_NOT_SUPPORTED_** if the audio resource is not available for tone playing.
- o **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note:

An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a tone is stopped automatically at the end of the duration time.

- Example

```
// audio resource handle
  s32 handle;

  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {
            switch ( Event)
      {
          case ADL_AUDIO_EVENT_NORMAL_STOP :
               TRACE (( 1, " Audio handle %d : stop ", audioHandle ));

              // unsubscribe to the speaker
              Ret = adl_audioUnsubscribe ( handle );
              break;

          case ADL_AUDIO_EVENT_RESOURCE_RELEASED :
              // ...
          break;

          default : break;
      }
      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandle,
      ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a single tone
      Ret = adl_audioTonePlay( handle, 300, -10, 0, 0, 50 );
  }
```

3.33.7.2 The adl_audioDTMFPlay Function

This function allows a DTMF tone to be played on the current speaker or on voice call TX (in communication only).

It is not possible to play DTMF on the buzzer. The specified output stops to play at the end of tone duration or on an application request (refer to **adl_audioStop** for more information, section 3.33.9.1). Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface User Guide [1] for more information.

- Prototype

```
s32 adl_audioDTMFPlay   (s32          audioHandle,
                          ascii        DTMF,
                          s8           Gain,
                          u32          Duration );
```

- Parameters

audioHandle

Handle of the audio resource which will play DTMF tone (current speaker or voice call TX).

DTMF

DTMF to play (0-9,A-D,*,#).

Gain

This parameter sets the tone gain (dB), and is only for the speaker.

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
  - **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for DTMF playing.
  - **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for DTMF playing.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Notes:

- An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a DTMF is stopped automatically at the end of the duration time.
- A DTMF can not be stopped on client request when DTMF is played on voice call TX.

* Example

```
// audio resource handle
s32 handle;

  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {

      switch ( Event)
      {
          case ADL_AUDIO_EVENT_NORMAL_STOP :
              TRACE (( 1, " Audio handle %d : stop ", audioHandle ));

              // unsubscribe to the current speaker
              Ret = adl_audioUnsubscribe ( handle );
          break;

          case ADL_AUDIO_EVENT_RESOURCE_RELEASED :
              // ...
          break;

          default : break;
      }
      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
     handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandler,
     ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a DTMF tone
      Ret = adl_audioDTMFPlay( handle, 'A', -10, 10);
  }
```

3.33.7.3 The adl_audioMelodyPlay Function

This function allows to play a defined melody on current speaker or buzzer. The specified output stops the playing process on an application request (refer to **adl_audioStop** for more information, section 3.33.9.1) or when the melody has been played the same number of time than that is specified in CycleNumber. Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface User Guide [1] for more information.

- Prototype

```
s32 adl_audioMelodyPlay (s32            audioHandle,
                         u16 *          MelodySeq,
                         u8             Tempo,
                         u8             CycleNumber,
                         s8             Gain  );
```

- Parameters

audioHandle

Handle of the audio resource which will play Melody (current speaker or buzzer).

MelodySeq

Melody to play. A melody is defined by an u16 table , where each element defines a note event, duration and sound definition. The melody sequence has to finish by a NULL value. (refer to 3.33.3.2 **ADL_AUDIO_NOTE_DEF** for more information)

Tempo

Tempo is defined in bpm (1 beat = 1 quarter note).

CycleNumber

Number of times the melody should be played. If not specified, the cycle number is infinite, Melody should be stopped by client.

Gain

This parameter sets melody gain (dB).

- Returned values
  - o **OK** on success
  - o **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - o **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
  - o **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for melody playing.
  - o **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for melody playing.
  - o **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note:

An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a Melody is stopped automatically at the end of the cycle number.

- Example

```
// audio resource handle
  s32 handle;

// Melody buffer
u16*MyMelody={ADL_AUDIO_NOTE_DEF( ADL_AUDIO_A,3,ADL_AUDIO_DOTTED_QUARTER),
              ADL_AUDIO_NOTE_DEF( ADL_AUDIO_CS,5,ADL_AUDIO_DOTTED_HALF),
              ADL_AUDIO_NOTE_DEF( ADL_AUDIO_E,1,ADL_AUDIO_WHOLE_NOTE ),
              ... ,
              ADL_AUDIO_NOTE_DEF( ADL_AUDIO_AS,3,ADL_AUDIO_HEIGHTH),
              0 };

// audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {
      s32 Ret;

      switch ( Event)
      {
          case ADL_AUDIO_EVENT_NORMAL_STOP :
              TRACE (( 1, " Audio handle %d : stop ", audioHandle ));

              // unsubscribe to the buzzer
              Ret = adl_audioUnsubscribe ( handle );

          break;

          case ADL_AUDIO_EVENT_RESOURCE_RELEASED :
              // ...
          break;

          default : break;
      }
      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_BUZZER, MyAudioEventHandler ,
ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a Melody
      Ret = adl_audioMelodyPlay( handle, MyMelody, 10, 2, -10);
  }
```

3.33.7.4 The adl_audioTonePlayExt Function

This function plays a single or dual tone on current speaker and only a single tone on buzzer.
Only the speaker output is able to play tones in two frequencies. The second tone parameters are ignored on buzzer output.
The specified output stops to play at the end of tone duration or on an application request (refer to **adl_audioStop** for more information section 3.33.9.1).
Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT commands Interface User Guide [1] for more information.

- Prototype

```
s32 adl_audioTonePlayExt(s32        audioHandle,
                         u16        Frequency1,
                         s16        Gain1,
                         u16        Frequency2,
                         s16        Gain2,
                         u32        Duration );
```

- Parameters

audioHandle

Handle of the audio resource which will play tone (current speaker or buzzer).

Frequency1

Frequency for the 1st tone (Hz).

Gain1

This parameter sets the tone gain which will be applied to the 1st frequency value (unit: 1/100 of dB).

Frequency2

Frequency for the 2nd tone (Hz), only processed on current speaker. Frequency2 has to set to 0 to play a single tone on current speaker.

Gain2

This parameter sets the tone gain which will be applied to the 2nd frequency value (unit : 1/100 of dB).

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.

- o **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for tone playing.
- o **ADL_RET_ERR_NOT_SUPPORTED_** if the audio resource is not available for tone playing.
- o **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Note:

An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a tone is stopped automatically at the end of the duration time.

3.33.7.5 The adl_audioDTMFPlayExt Function

This function allows a DTMF tone to be played on the current speaker or on voice call TX (in communication only).

It is not possible to play DTMF on the buzzer.

The specified output stops to play at the end of tone duration or on an application request (refer to **adl_audioStop** for more information, section 3.33.9.1). Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface User Guide [1] for more information.

- Prototype

```
s32 adl_audioDTMFPlayExt(s32        audioHandle,
                         ascii      DTMF,
                         s16        Gain,
                         u32        Duration );
```

- Parameters

audioHandle

Handle of the audio resource which will play DTMF tone (current speaker or voice call TX).

DTMF

DTMF to play (0-9,A-D,*,#).

Gain

This parameter sets the tone gain (unit: 1/100 of dB), and is only for the speaker.

Duration

This parameter sets the tone duration (ms). The value has to be a 20-ms multiple.

- Returned values
  - o **OK** on success
  - o **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - o **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - o **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.

- **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for DTMF playing.
- **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for DTMF playing.
- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Notes:

- An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a DTMF is stopped automatically at the end of the duration time.
- A DTMF cannot be stopped on client request when DTMF is played on voice call TX.

3.33.7.6 The adl_audioMelodyPlayExt Function

This function allows to play a defined melody on current speaker or buzzer.

The specified output stops the playing process on an application request (refer to **adl_audioStop** for more information, section 3.33.9.1) or when the melody has been played the same number of time than that is specified in CycleNumber.

Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Commands Interface User Guide [1] for more information.

- Prototype

```
s32 adl_audioMelodyPlayExt (s32          audioHandle,
                            u16 *        MelodySeq,
                            u8           Tempo,
                            u8           CycleNumber,
                            s16          Gain  );
```

- Parameters

audioHandle

Handle of the audio resource which will play Melody (current speaker or buzzer).

MelodySeq

Melody to play. A melody is defined by an u16 table , where each element defines a note event, duration and sound definition. The melody sequence has to finish by a NULL value. (refer to 3.33.3.2 **ADL_AUDIO_NOTE_DEF** for more information)

Tempo

Tempo is defined in bpm (1 beat = 1 quarter note).

CycleNumber

Number of times the melody should be played. If not specified, the cycle number is infinite; Melody should be stopped by client.

Gain

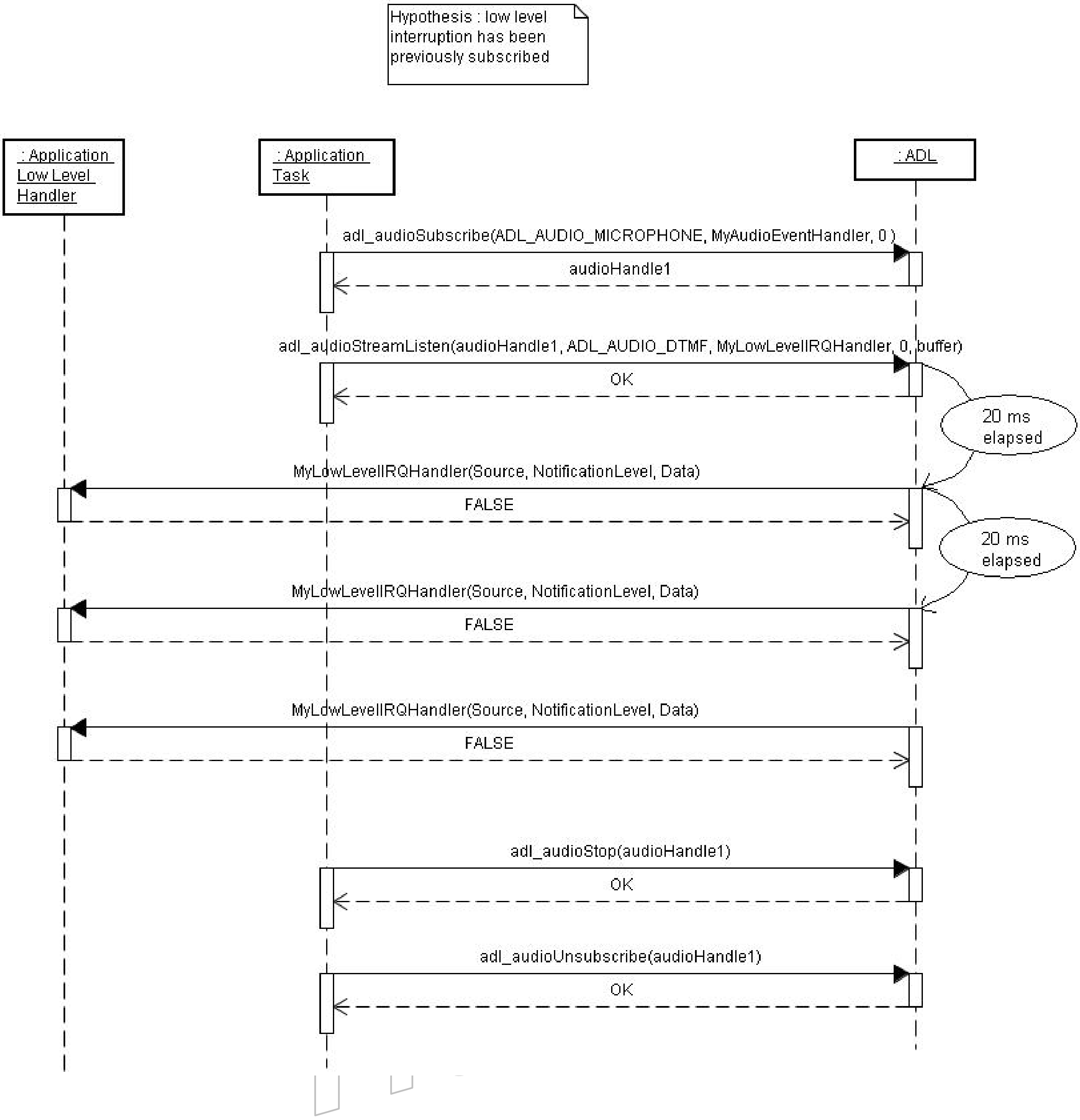
This parameter sets melody gain (unit: 1/100 of dB).

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
  - **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for melody playing.
  - **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for melody playing.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

  Note:
  - An event **ADL_AUDIO_EVENT_NORMAL_STOP** is sent to the owner resource when a Melody is stopped automatically at the end of the cycle number.

3.33.8 Audio stream

These functions allows to play or listen an audio stream.



3.33.8.1 The adl_audioStreamPlay Function

This function allows to play an audio sample stream on the current speaker or on voice call TX.

Playing an audio sample stream implies that low level interrupt and/or high level interrupt have been previously subscribed

(Refer to 3.25.10 **adl_irqSubscribe** in ADL user guide for more information).

Moreover, memory space has to be allocated for the audio stream buffer before playing starts and it has to be released after playing stops.

Only audio PCM sample can be played.

Use **adl_audioGetOption** function to obtain the parameters range. (also refer to AT Commands User Guide [1] for more information).

(Refer to **adl_audioStreamDataBuffer_u::PCMData** to get information about the data buffer format).

- Prototype

```
s32 adl_audioStreamPlay   (s32                  audioHandle,
                           adl_audioFormats_e   audioFormat,
                           s32                  LowLevelIRQHandle,
                           s32                  HighLevelIRQHandle,
                           void *               buffer );
```

- Parameters

  audioHandle

  Handle of the audio resource which will play audio stream (current speaker or voice call TX).

  audioFormat

  Stream audio format. Only **ADL_AUDIO_PCM_MONO_8K_16B** format is available to be played (Refer to **adl_audioFormats_e** for more information, section 3.33.4.3).

  LowLevelIRQHandle

  Low level IRQ handle previously returned by IRQ subscription (please refer to **adl_irqSubscribe**, section 3.25.10, for more information).

  HighLevelIRQHandle

  High level IRQ handle previously returned by IRQ subscription (please refer to **adl_irqSubscribe**, section 3.25.10, for more information).

  buffer

  contains sample to play.

- Returned values

  - **OK** on success
  - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.

- o **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
- o **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio pre-defined signal is playing on the required audio resource.
- o **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for audio stream playing or if interrupt handler identifiers are invalid.
- o **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for audio stream playing.
- o **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

Notes:

- o To work properly, LowLevelIRQHandle is mandatory. The low level interrupt has to be previously subscribed with **ADL_IRQ_OPTION_AUTO_READ** option.
- o The HighLevelIRQHandle is optional.
- o Each time an audio sample is required, an interrupt handler will be notified to send the data. The interrupt identifier will be set to **ADL_IRQ_ID_AUDIO_RX_PLAY** or **ADL_IRQ_ID_AUDIO_TX_PLAY**, according to the resource used to start the stream play.
- o Some audio filters will be deactivated for audio sample playing (refer to "audio command" chapter in the AT command Interface Guide [1] for more information).
- o For audio interrupt subscription **ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT** option is not available.

- Example

```
// audio resource handle
s32 handle;

// audio stream buffer
void * StreamBuffer;

// PCM samples
u16 PCM_Samples[160] = { ... , ... , ... , ... , ... , ... , 0 };   //
size of PCM sample = 320 bytes

// PCM samples index
u8 indexPCM = 0;

// Low level interrupt handler
bool MyLowLevelIRQHandler ( adl_irqID_e Source, adl_irqNotificationLevel_e
Notification Level, adl_irqEventData_t * Data )
{
    // copy PCM sample to play
    wm_strcpy( StreamBuffer, PCM_Samples );
    // Set BufferReady flag to TRUE
    *( ( adl_audioStream_t * )Data->SourceData )->BufferReady = TRUE;

    //...

    return FALSE;
}
```

```
// audio event call-back function
void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
{

    // ...

    return;
}

void adl_main ( adl_InitType_e InitType )
{
    s32 Ret;
    s32 BufferSize;

    // Subscribe to the current speaker
    handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandler ,
    ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

    // Memory allocation
    Ret = adl_audioGetOption ( handle,
    ADL_AUDIO_PCM_8K_16B_MONO_BUFFER_SIZE, &BufferSize )
    StreamBuffer = adl_memGet( BufferSize );   // release memory after
                                                  audio stream playing

    // Play an audio PCM stream
    Ret = adl_audioStreamPlay( handle, ADL_AUDIO_PCM_MONO_8K_16B
    MyLowLevelIRQHandler, 0, StreamBuffer);
}
```

3.33.8.2 The adl_audioStreamListen Function

This function allows listening to a DTMF tone or an audio sample from microphone or voice call RX.

Listening to an audio sample stream implies that low level interrupt and/or high level interrupt have been previously subscribed (refer to **adl_irqSubscribe** for more information, section 3.25.10).

Moreover, memory space has to be allocated for the audio stream buffer before listening starts and it has to be released after listening stops. Use **adl_audioGetOption** function to obtain the parameters range. Please also refer to AT Command Interface Guide for more information. According to audio format stream and DTMF decoding mode (for listening to an audio DTMF stream), buffer has a different format:

- for listening to an audio sample, **_adl_audioStreamDataBuffer_u**::**PCMData** format is used.
- for listening to a DTMF stream, in Raw mode (refer to **ADL_AUDIO_DTMF_DETECT_BLANK_DURATION** for more information about Raw mode, section 3.33.4.5), **_adl_audioDecodedDtmf_u::DecodedDTMFChars** format is used.

- for listening to a DTMF stream, in no Raw mode (refer to 3.33.4.5 **ADL_AUDIO_DTMF_DETECT_BLANK_DURATION** for more information about no Raw mode), **adl_audioPostProcessedDecoder_t** structure is used.

- Prototype

```
s32 adl_audioStreamListen (s32                  audioHandle,
                           adl_audioFormats_e   audioFormat,
                           s32                  LowLevelIRQHandle,
                           s32                  HighLevelIRQHandle,
                           void *               buffer );
```

- Parameters

audioHandle

Handle of the audio resource from which to listen the audio stream (microphone or voice call RX).

audioFormat

Stream audio format (refer to **adl_audioFormats_e** for more information, section 3.33.4.3).

LowLevelIRQHandle

Low level IRQ handle previously returned by IRQ subscription (please refer to **adl_irqSubscribe**, section 3.25.10, for more information).

HighLevelIRQHandle

High level IRQ handle previously returned by IRQ subscription (please refer to **adl_irqSubscribe**, section 3.25.10, for more information).

buffer

contains received decoded DTMF or audio samples.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - **ADL_RET_ERR_BAD_STATE** if an audio stream is listening or audio signal is playing on the required audio resource.
  - **ADL_RET_ERR_BAD_HDL** if the audio resource is not allowed for audio stream listening or if interrupt handler identifiers are invalid.
  - **ADL_RET_ERR_NOT_SUPPORTED** if the audio resource is not available for audio stream listening.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

<u>Notes:</u>

- The **LowLevelIRQHandle** is optional if the **HighLevelIRQHandle** is supplied.
- The **HighLevelIRQHandle** is optional if the **LowLevelIRQHandle** is supplied.
- Each time an audio sample or DTMF sequence is detected, an interrupt handler will be notified to require the data. The interrupt identifier will

be set to **ADL_IRQ_ID_AUDIO_RX_LISTEN** or **ADL_IRQ_ID_AUDIO_TX_LISTEN**, according to the resource used to start the stream listen.

- o All audio filters will be deactivated for DTMF listening and only some audio filters for audio sample listening (refer to "audio command" chapter in the AT command Interface Guide [1] for more information).
- o For audio interrupt subscription, **ADL_IRQ_OPTION_POST_ACKNOWLEDGEMENT** option is not available.

- Example

```
// audio resource handle
  s32 handle;

  // audio stream buffer
  void * StreamBuffer;

  // Low level interrupt handler
  bool MyLowLevelIRQHandler ( adl_irqID_e Source, adl_irqNotificationLevel_e
  Notification Level, adl_irqEventData_t * Data )
  {
      TRACE (( 1, "DTMF received : %c, %c ", StreamBuffer[0],
StreamBuffer[1] ));

      return FALSE;
  }

  // audio event call-back function
  void MyAudioEventHandler ( s32 audioHandle, adl_audioEvents_e Event )
  {

      // ...

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;;
       s32 BufferSize

      // Subscribe to the current microphone
      handle = adl_audioSubscribe ( ADL_AUDIO_MICROPHONE,
      MyAudioEventHandler , ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Memory allocation
      Ret = adl_audioGetOption (handle, ADL_AUDIO_PCM_8K_MONO_BUFFER_SIZE,
&BufferSize )
   StreamBuffer = adl_memGet( BufferSize);   // release memory after audio
                                                  stream listening

      // Listen to audio DTMF stream
      Ret = adl_audioStreamListen( handle, ADL_AUDIO_DTMF
      MyLowLevelIRQHandler, 0, StreamBuffer);
  }
```

3.33.9 Stop

3.33.9.1 The adl_audioStop Function

This function allows to:

- stop playing a tone on the current speaker or on the buzzer,
- stop playing a DTMF on the current speaker or on the voice call TX,
- stop playing a melody on the current speaker or on the buzzer,
- stop playing an audio PCM stream on the current speaker or on the voice call TX,
- stop listening to an audio DTMF stream from current microphone or voice call RX,
- stop listening to an audio sample stream from current microphone or voice call RX.

ADL_AUDIO_EVENT_NORMAL_STOP event will not be sent to application.

- Prototype

```
s32 adl_audioStop (s32       audioHandle );
```

- Parameters

  audioHandle

  Handle of the audio resource which has to stop its process.

- Returned values
  - **OK** on success.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.
  - **ADL_RET_ERR_BAD_STATE** if no audio process is running on the required audio resource.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

- Example

```
// audio resource handle
  s32 handle;

  void adl_main ( adl_InitType_e InitType )
  {
      s32 Ret;

      // Subscribe to the current speaker
      handle = adl_audioSubscribe ( ADL_AUDIO_SPEAKER, MyAudioEventHandler ,
      ADL_AUDIO_RESOURCE_OPTION_FORBID_PREEMPTION );

      // Play a single tone
      Ret = adl_audioTonePlay( handle, 300, -10, 0, 0, 50 );

      // Stop playing the single tone
      Ret = adl_audioStop( handle );

      // unsubscribe to the current speaker
      Ret = adl_audioUnsubscribe ( handle );

  }
```

3.33.10 Set/Get options

3.33.10.1 The adl_audioSetOption Function

This function allows to set an audio option according to audio resource and option type specified. Several option types are only readable, so this function cannot be used with them (refer to 3.33.4.5 **adl_audioOptionTypes_e** for more information).

- Prototype

```
s32 adl_audioSetOption (s32                     audioHandle,
                        adl_audioOptionTypes_e  audioOption,
                        s32                     value   );
```

- Parameters

  audioHandle

  Handle of the audio resource.

  audioOption

  This parameter defines audio option to set (refer to 3.33.4.5 **adl_audioOptionTypes_e** for more information).

  value

  Defines setting value for option.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.

3.33.10.2 The adl_audioGetOption Function

This functions allows to get information about audio service according to audio resource and option type specified.

- Prototype

```
s32 adl_audioGetOption (s32                    audioHandle,
                        adl_audioOptionTypes_e  audioOption,
                        s32 *                   value );
```

- Parameters

  audioHandle

  Handle of the audio resource.

  audioOption

  audio option which wishes to get information (refer to 3.33.4.5 **adl_audioOptionTypes_e** for more information).

  value

  option value according to audio option which has been set.

- Returned values
  - 0 value option value according to audio option which has been set.
  - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
  - **ADL_RET_ERR_UNKNOWN_HDL** if the provided handle is unknown.

3.34 ADL Secure Data Storage Service

The ADL supplies Secure Data Storage Service interface to

- read/write/query data stored in ciphered format in non volatile memory,
- update cryptographic keys in order to block replay/re-download attacks.

The defined operations are:

- An **adl_sdsWrite** function to write secured data.
- An **adl_sdsRead** function to read secured data.
- An **adl_sdsQuery** function to require size of one of secured entries.
- An **adl_sdsDelete** function to delete one of secured entries.
- An **adl_sdsStats** function to get statistics about secured data storage.
- An **adl_sdsUpdateKeys** function to update the cryptographic keys.

Note:

These functions are available only if:

- they are used with a compatible platform.
- the Secured Data Storage feature is properly activated on the production line
- the objects are not erased, otherwise Wireless CPU® has to be returned in production line

Otherwise, every function cited above will return the error code **ADL_RET_ERR_NOT_SUPPORTED**.

3.34.1 Required Header File

The header file for the functions dealing with the ADL Secure Data Storage Service public interface is:

adl_sds.h

3.34.2 Data Structure

3.34.2.1 The adl_sdsStats_t Structure

Data storage statistics contains information about secured data storage. It has to be used with adl_sdsStats API. .

- Code

```
typedef struct
{
  u32   FreeSpace
  u32   TotalSpace
  u16   EntryCount
  u16   MaxEntry
  u32   MaxEntrySize
}adl_sdsStats_t;
```

- Description

FreeSpace

Available space for secured entries.

Caution: This figure does not depend only on written data but depends on the state of the underlying storage media too. It might increase or decrease as data entries sharing the same space as ciphered entries are created or deleted.

TotalSpace

Total space allocated for ciphered entries. This figure is a quota, and must be treated as such. Because ciphered entries share storage media with other information, this quota might be unaccessible if, for example, the underlying storage medium is near its full capacity.

EntryCount

Total number of secured entries.

MaxEntry

Maximal number of secured entry.

Note: The max number of secured entries depends on the underlying storage service. There might be less available entries if this storage service is near its maximum capacity.

MaxEntrySize

Maximal size of one secured entry. It's defined in number of bytes.

3.34.3 Defines

3.34.3.1 ADL_SDS_RET_ERR_ENTRY_NOT_EXIST

Entry does not exist.

```
#define ADL_SDS_RET_ERR_ENTRY_NOT_EXIST   ADL_RET_ERR_SPECIFIC_BASE
```

3.34.3.2 ADL_SDS_RET_ERR_MEM_FULL

Not enough space memory to write secured data.

```
#define ADL_SDS_RET_ERR_MEM_FULL       ADL_RET_ERR_SPECIFIC_BASE - 1
```

3.34.4 The adl_sdsWrite Function

This function allows to store data in a secured entry, data are ciphered. This function creates a new entry or updates an existing one.

- Prototype

```
s32 adl_sdsWrite     ( u32                ID,
                       u32                Length,
                       void *             Source );
```

- Parameters

  ID:

  Numeric ID of the entry. The ID range is from 0 to MaxEntry (returned by **adl_sdsStats**). Refer to **adl_sdsStats_t** to get more information about MaxEntry.

  Length

  Size of the data to write in the entry. Use **adl_sdsStats** to get the maximum size for one secured entry (refer to MaxEntrySize in **adl_sdsStats_t** to get more information).

  Source

  Pointer to the source buffer. It contains data to write.

- Returned values
  - **OK** on success
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
    - **ADL_SDS_RET_ERR_MEM_FULL** no enough memory is available for writing.
    - **ADL_RET_ERR_NOT_SUPPORTED** writing operation is not available.
    - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.34.5 The adl_sdsRead Function

This function allows to retrieve data from a secured entry. Data which has been previously written with **adl_sdsWrite** API.

- Prototype

```
s32 adl_sdsRead      (  u32              ID,
                        u32              Offset,
                        u32              Length,
                        void *           Destination );
```

- Parameters

ID:

Numeric ID of the entry. The ID range is from 0 to MaxEntry (returned by **adl_sdsStats**). Refer to **adl_sdsStats_t** to get more information about MaxEntry.

Offset

Offset in the secured entry, defined in number of bytes. It allows to retrieve a part of the entry. It is an offset in relation to the first byte of the entry.

Length

Size of data to read in the secured entry. Use **adl_sdsQuery** API to get the maximal length for the required entry.

Destination

Pointer to the destination buffer. It contains data to retrieve.

- Returned values
    - **OK** on success
    - A negative error value otherwise:
        - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
        - **ADL_SDS_RET_ERR_ENTRY_NOT_EXIST** if entry ID does not exist.
        - **ADL_RET_ERR_NOT_SUPPORTED** reading operation is not available.
        - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.34.6 The adl_sdsQuery Function

This function allows to check if a secured entry exists and gets its size.

- Prototype

```
s32 adl_sdsQuery     (  u32              ID,
                        u32*             Length );
```

- Parameters

  ID:

  Numeric ID of the entry. The ID range is from 0 to MaxEntry (returned by **adl_sdsStats**). Refer to **adl_sdsStats_t** to get more information about MaxEntry.

  Length

  Output pointer for the entry size. It can be set to NULL.

- Returned values
  - **OK** on success
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
    - **ADL_SDS_RET_ERR_ENTRY_NOT_EXIST** if entry ID does not exist.
    - **ADL_RET_ERR_NOT_SUPPORTED** operation is not available.
    - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.34.7 The adl_sdsDelete Function

This function allows to delete one of existing entries.

- Prototype

```
s32 adl_sdsDelete    ( u32          ID);
```

- Parameters

  ID:

  Numeric ID of the entry. The ID range is from 0 to MaxEntry (returned by **adl_sdsStats**). Refer to **adl_sdsStats_t** to get more information about MaxEntry.

- Returned values
  - **OK** on success
  - A negative error value otherwise:
    - **ADL_RET_ERR_PARAM** if parameters have an incorrect value or secured entry doesn't exist.
    - **ADL_SDS_RET_ERR_ENTRY_NOT_EXIST** if entry ID does not exist.
    - **ADL_RET_ERR_NOT_SUPPORTED** deletion operation is not available.
    - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.34.8 The adl_sdsStats Function

This function allows to retrieve information about secured data storage as free memory space or total memory space.

- Prototype

```
s32 adl_sdsStats (adl_sdsStats*          Stats);
```

- Parameters

Stats:

Pointer on statistical information of secured data storage. (refer to **adl_sdsStats_t** to have more information about statistics).

- Returned values
  - o **OK** on success
  - o A negative error value otherwise:
    - ▪ **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
    - ▪ **ADL_RET_ERR_NOT_SUPPORTED** operation is not available.
    - ▪ **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.34.9 The adl_sdsUpdateKeys Function

This function allows to re-generate the internal cryptographic keys. This function has to be used to defeat possible replay or re-download attacks. Once the keys are re-generated, all the stored data remain available and still readable by application, but the processor will not be able to re-use a previous image of the non-volatile memory with old cryptographic keys.

- Prototype

```
s32 adl_sdsUpdateKeys ( void );
```

Note:

This function is synchronous and its exectution time is independent of the number of entries.

Warning:

This must be used with caution because of the limited life expectancy of the non-volatile memory implied in this process. For example, a WMP100 processor can, at most, withstand 2x10^6 key changes: changing them every second would therefore wear out the processor after 1.5 year.

- Returned values
  - o **OK** on success
  - o A negative error value otherwise:
    - ▪ **ADL_RET_ERR_PARAM** if parameters have an incorrect value.
    - ▪ **ADL_RET_ERR_NOT_SUPPORTED** updating operation is not available.
    - ▪ **ADL_RET_ERR_FATAL** EEPROM cannot be written.

- **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.34.10 Example

The code sample below illustrates a nominal use case of the ADL Secure Data Storage Service public interface (error cases are not handled).

```
// ...
   // decrement counter
   u32 n=10;
   u32 size;
   u32 offset=0;
   adl_sdsWrite( COUNTER_ID, offset, sizeof(u32), &n );
   adl_sdsQuery( COUNTER_ID, &size );
   adl_sdsRead( COUNTER_ID, offset, size, &n );
   n--;
   adl_sdsWrite( COUNTER_ID, size, &n );

   // ensure that from now on, any previously
   // stored memory image becomes incompatible
   // with this processor
   adl_sdsUpdateKeys();
   // ...

   adl_sdsRead( COUNTER_ID, offset, sizeof(u32), &n );
   // delete entry
   adl_sdsDelete( COUNTER_ID );
```

3.35 ADL WatchDog Service

ADL provides a watchdog service to access to the Wireless CPU®s WatchDog.

Note: the timing unit is a tick which corresponds to 18.5 ms.

- Hardware watchdog put to sleep

Because an application may launch heavy treatments that can take more than the hardware watchdog duration (one minute for example) and because the watchdog cannot be stopped once it had been started, system provides a way to deactivate the hardware watchdog from the application point of view for a given time. In fact, during this time, system rearms by itself the hardware watchdog application in a high priority task because the IDLE task cannot take the focus while the application treatments are not finished.

The defined operations are:

  - A **adl_wdPut2Sleep**
  - A **adl_wdAwake**

- Application watchdog Management

Application watchdog can be activated with a given duration. Once the application watchdog is activated, the application binary has to rearm regularly the application watchdog to indicate that it is still alive. Else, a back trace is generated and a reset occurs. Application watchdog can be deactivated or reactivated with a new duration.

The defined operations are:

  - A **adl_wdRearmAppWd**
  - A **adl_wdActiveAppWd**
  - A **adl_wdDeActiveAppWd**

3.35.1 Required Header File

The header file for the functions dealing with the ADL WatchDog Service public interface is:

adl_wd.h

3.35.2 The adl_wdPut2Sleep Function

This function enables to launch an automatic hardware watchdog relaunch for a given duration. Thanks to this function, during the watchdog sleep duration, application treatments can take more than hardware watchdog duration even if IDLE task cannot have the CPU focus for more than hardware watchdog duration. Once the sleep duration expired, the IDLE task must receive back the CPU focus in less than the hardware watchdog duration, else a watchdog reset occurs.

Note:

This must be called just before an heavy treatment to avoid watchdog reset. The argument has to be strictly positive.

- Prototype

  `u32 adl_wdPut2Sleep ( u32 i_u32_SleepDuration );`

- Parameters

  i_u32_SleepDuration:

  Watchdog sleep duration in number of ticks (timer macro **ADL_TMR_S_TO_TICK(SecT)** - can be used for duration conversion).

- Returned values
  - o **OK** or **ADL_RET_ERR_PARAM** if wrong argument.

3.35.3 The adl_wdAwake Function

The adl_wdAwake function enables to cancel watchdog inactivation.

Note:

This should be called just after an heavy treatment if watchdog had been inactivated to force the restore of default behavior. If not called, default behavior will be restored automatically at the expiration of watchdog sleep duration.

- Prototype

  `u32 adl_wdAwake ( void );`

- Returned values

  Remaining time before automatic watchdog reactivation in number of ticks.

3.35.4 Example

Here is an example of how to use the watchdog API access functions.

```
void CallMyHeavyTreatpments(void)
  {
     // To store remaining time before the end of watchdog inactivation
     u32 i_u32_ReaminingTime;

     // Watchdog inactivation for 30 seconds
     adl_wdPut2Sleep(ADL_TMR_S_TO_TICK(30));

     // Launch heavy treatment
     MyHeavyTreatemnt();

     // Watchdog reactivation
     i_u32_ReaminingTime = adl_wdAwake();

     printf("Watchdog is to be awaken in %d number of ticks",
     i_u32_ReaminingTime );
  }
```

3.35.5 The adl_wdRearmAppWd Function

Enable to rearm the application watchdog with the stored watchdog duration.

Note:

Application can use a cyclic timer to regularly rearm the application watchdog.

OK is returned and nothing happens if **adl_wdActiveAppWd** has not been called before.

- Prototype

```
s32 adl_wdRearmAppWd   ( void );
```

- Returned values

  OK or **ADL_RET_ERR_NOT_SUPPORTED** if watchdog service not supported.

3.35.6 The adl_wdActiveAppWd Function

Once started application watchdog must be rearmed regularly (no matter how) to indicate that it is still alive. If the watchdog timer expired, the hardware watchdog will not be rearmed anymore and the Wireless CPU®s will reset.

- Prototype

```
s32 adl_wdActiveAppWd   (u32      i_u32_Duration );
```

Note:

Argument has to be strictly positive.

- Parameters

  i_u32_Duration:

  Software application watchdog duration in number of ticks (timer macro ADL_TMR_S_TO_TICK(SecT) - can be used for duration conversion).

- Returned values

  OK or **ADL_RET_ERR_NOT_SUPPORTED** if watchdog service not supported.

3.35.7 The adl_wdDeActiveAppWd Function

The **adl_wdDeActiveAppWd** function enables to stop watchdog.

Note:

OK is returned and nothing happens if **adl_wdActiveAppWd** has not been called before.

- Prototype

```
s32 adl_wdDeActiveAppWd ( void );
```

- Returned values

  OK or **ADL_RET_ERR_NOT_SUPPORTED** if watchdog service not supported.

3.35.8 Example

Here is an example of how to use the application watchdog API access functions.

```
void CallMyHeavyAppliTreatpments(void)
{
   adl_tmr_t *tt;

   // Lets activate the application watchdog for 30 seconds
   adl_wdActiveAppWd(ADL_TMR_S_TO_TICK(30));

   // Lets suscribe to a 25 sec timer
   tt = (adl_tmr_t *)adl_tmrSubscribe (TRUE,
                                       25,
                                       ADL_TMR_TYPE_100MS,
                                       (adl_tmrHandler_t)Timer_Handler);

   // Launch heavy appli treatment
   MyHeavyAppliTreatemnt();
}

void Timer_Handler( u8 Id )
{
   if ( (process has not ended)
   {
       if (there is some activities)
       {
           // Rearm the application watchdog for another go
           adl_wdRearmAppWd();
       }
       else
       {
           // the process has not ended and there is no activities ->
             application watchdog reset
       }
   }
   else // process has ended
   {
       // the process has ended we can now deactivate the application
          watchdog
       adl_wdDeActiveAppWd();
   }
}
```

3.36 ADL Layer 3 Service

The ADL supplies Layer3 Service interface allows to get information about Layer 3 as PLMN scan information.

The defined operations are:

- A **adl_L3infoSubscribe** function to subscribe to the L3 information service
- A **adl_L3infoUnsubscribe** function to unsubscribe to the L3 information service.

3.36.1 Required Header File

The header file for the functions dealing with the ADL Layer 3 Service public interface is:

```
adl.L3info.h
```

3.36.2 The adl_L3InfoChannelList_e

List of information channel which are available.

- Code

```
typedef enum
{
  adl_L3infoChannelList_e     ADL_L3INFO_SCAN
}adl_L3infoChannelList_e;
```

- Description

ADL_L3INFO_SCAN

This channel allows to retrieve information about PLMN Scan:

- o power min, max, average
- o cell synchronization refer to [2] file to have more details about information structure which are returned by Scan channel

3.36.3 The Layer3 infoEvent Handler

Such a call-back function has to be supplied to ADL through the adl_L3infoSubscribe interface in order to receive L3 information according to channels and related events.

- Prototype

```
typedef void(*)adl_L3infoEventHandler_f(u32                        Time,
                                        adl_L3infoChannelList_e ChannelId,
                                        u32                        EventId,
                                        u32                        Length,
                                        void *                     Info );
```

- Parameters

  Time

  Reserved for Future Use.

  ChannelId

  Channel identity which provides information. (refer to **adl_L3infoChannelList_e** for more information).

  EventId

  Event identity. refer to [2] for more information about possible event.

  Length

  Length of "Info" content.

  Info

  Information content according to ChannelID and EventID. Refer to [2] for more information about the type of "Info".

3.36.4 The adl_L3infoSubscribe Function

This function allows to subscribe to one of the available information channel of the Layer 3.

A call-back function is associated for Layer 3 events. It allows to retrieve information relative to the channel requested.

- Prototype

```
s32 adl_L3infoSubscribe    ( adl_L3infoChannelList_e    ChannelId,
                             adl_L3infoEventHandler_f   L3infoHandler );
```

- Parameters

  ChannelId

  Information channel requested.(refer to **adl_L3infoChannelList_e** for more information).

  L3infoHandler

  Application provides Layer 3 event call-back function (refer to **adl_L3infoEventHandler_f** for more information).

- Returned values
  - Positive or NULL if allocation succeed, returns handle which has to be used on further L3 info API functions calls
  - **ADL_RET_ERR_PARAM** if parameter has an incorrect value.
  - **ADL_RET_ERR_ALREADY_SUBSCRIBED** if the channel information is already subscribed.
  - **ADL_RET_ERR_NOT_SUPPORTED** If the Raw Spectrum Information feature is not enabled on the Wireless CPU®.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.36.5 The adl_L3infoUnsubscribe Function

This function allows to unsubscribe to the specific channel L3 information flow which has been subscribed previously with **adl_L3infoSubscribe** function.

- Prototype

  `s32 adl_L3infoUnsubscribe ( u32 Handle );`

- Parameters

  Handle

  handle previously returned by **adl_L3infoSubscribe** function.

- Returned values
  - **OK** on success
  - **ADL_RET_ERR_UNKNOWN_HANDLE** if the provided handle is unknown.
  - **ADL_RET_ERR_SERVICE_LOCKED** if called from a low level interrupt handler.

3.36.6 Example

These function allows to subscribe or unsubscribe to one of information channel available from Layer 3.

```
// Channel info handle
  s32 handle;

  // info channel event call-back function
  void MyChannelEventHandler( u32 Time, adl_L3infoChannelList_e ChannelId,
  u32 EventId, u32 Length, void * Info )
  {

      switch ( EventId)
      {
          ...
      }

      adl_L3infoUnsubscribe( handle );

      return;
  }

  void adl_main ( adl_InitType_e InitType )
  {

      // Subscribe to PLMN Scan channel information
      handle = adl_L3infoSubscribe ( ADL_L3INFO_SCAN,
                                     MyChannelEventHandler);

  }
```

3.36.7 PLMN SCAN Information Channel Interface

This page describes events and associated data structure to provide information about PLMN SCAN procedure.

The PLMN Scan procedure is composed by the following steps :

- At first a power measurement on each supported frequency is done.
- Then if sufficent power (> noise power level(~ -105dBm)) is detected on one or more cells, cell synchronisation attempt is performed on these cells.

The PLMN scan procedure can be initiated by the Wireless CPU® itself, for initial PLMN selection or automatic PLMN reselection purposes, or can be initiated by user with AT+COPS command for instance.

3.36.7.1 Measurements Information [WM_L3_INFO_SCAN_PWR event]

The Measurement information are reported each time a power measurement is required on all frequencies.

The corresponding reported data are statistics on the low band, the high band and the low+high band.

The total number of cells with a power level greater than the noise power level is also reported.

3.36.7.2 Cell Synchronisation Information [WM_L3_INFO_SCAN_SYNC_CELL event]

The Cell Sychronisation information are reported when a cell synchronisation attempt was done during the PLMN Scan procedure and

- if the Wireless CPU® is not camped on a cell (the number of synchro failure is updated)
- if the Wireless CPU® has just camped on a cell (CellCamped flag set): no other WM_L3_INFO_SCAN_SYNC_CELL event is reported after.

3.36.7.3 The wm_l3info_Scan_PowerInfo_t Structure

Power Measurement Information structure.

- Code

```
typedef struct
{
  wm_I3info_Scan_PowerStat_t   Total,
  wm_I3info_Scan_PowerStat_t   LowBand,
  wm_I3info_Scan_PowerStat_t   HighBand,
  u16                          NumberOfCellAboveNoise,
  bool                         CellCamped
}wm_I3info_Scan_PowerInfo_t;
```

- Description

  Total

  Power Measurement statistics for all bands.

  LowBand

  Power Measurement statistics for the low band (GSM/850).

  HighBand

  Power Measurement statistics for the high band (DCS/PCS).

  NumberOfCellAboveNoise

  Number of cells with a power level greater than the noise's one.

  CellCamped

  TRUE if Wireless CPU® is camped on a cell, else FALSE.

3.36.7.4 The wm_l3info_Scan_PowerStat_t Structure

Power Measurement structure.

- Code

```
typedef struct
{
  u32   NbFreq
  u8    Min
  u8    Max
  u8    Mean
  u32   Variance
}wm_I3info_Scan_PowerStat_t;
```

- Description

  NbFreq

  Number of frequencies.

  Min

  Minimal power level detected.

  Max

  Maximal power level detected.

  Mean

  Mean power level.

  Variance

  Variance.

3.36.7.5 The wm_l3info_Scan_SynchroCellInfo_t Structure

Cell Synchronization Information structure.

This information is reported each time a cell synchronisation is unsucessfull and no cell has been already synchronised, or when a first cell is synchronized.

- Code

```
typedef struct
{
  u16       NbCellTriedInLowBand,
  u16       NbCellTriedInHighBand,
  bool      CellCamped
}wm_I3info_Scan_SynchroCellInfo_t;
```

- Description

  NbCellTriedInLowBand

  Number of tried cell in low band since the start of the scan.

  NbCellTriedInHighBand

  Number of tried cell in high band since the start of the scan.

  CellCamped

  TRUE if Wireless CPU® is camped on a cell, else FALSE.

4 Error Codes

4.1 General Error Codes

Error Code	Error Value	Description
OK	0	No error response
ERROR	-1	general error code
ADL_RET_ERR_PARAM	-2	parameter error
ADL_RET_ERR_UNKNOWN_HDL	-3	unknown handler / handle error
ADL_RET_ERR_ALREADY_SUBSCRIBED	-4	service already subscribed
ADL_RET_ERR_NOT_SUBSCRIBED	-5	service not subscribed
ADL_RET_ERR_FATAL	-6	fatal error
ADL_RET_ERR_BAD_HDL	-7	Bad handle
ADL_RET_ERR_BAD_STATE	-8	Bad state
ADL_RET_ERR_PIN_KO	-9	Bad PIN state
ADL_RET_ERR_NO_MORE_HANDLES	-10	The service subscription maximum capacity is reached
ADL_RET_ERR_DONE	-11	The required iterative process is now terminated
ADL_RET_ERR_OVERFLOW	-12	The required operation has exceeded the function capabilities
ADL_RET_ERR_NOT_SUPPORTED	-13	An option, required by the function, is not enabled on the Wireless CPU®, the function is not supported in this configuration
ADL_RET_ERR_NO_MORE_TIMERS	-14	The function requires a timer subscription, but no more timers are available
ADL_RET_ERR_NO_MORE_SEMAPHORES	-15	The function requires a semaphore allocation, but there are no more free resource
ADL_RET_ERR_SERVICE_LOCKED	-16	If the function was called from a low lewel interruption handler (the function is forbidden in this case)
ADL_RET_ERR_SPECIFIC_BASE	-20	Beginning of specific errors range

4.2 Specific FCM Service Error Codes

Error code	Error value
ADL_FCM_RET_ERROR_GSM_GPRS_ALREADY_OPENNED	ADL_RET_ERR_SPECIFIC_BASE
ADL_FCM_RET_ERR_WAIT_RESUME	ADL_RET_ERR_SPECIFIC_BASE-1
ADL_FCM_RET_OK_WAIT_RESUME	OK+1
ADL_FCM_RET_BUFFER_EMPTY	OK+2
ADL_FCM_RET_BUFFER_NOT_EMPTY	OK+3

4.3 Specific Flash Service Error Codes

Error Code	Error Value
ADL_FLH_RET_ERR_OBJ_NOT_EXIST	ADL_RET_ERR_SPECIFIC_BASE
ADL_FLH_RET_ERR_MEM_FULL	ADL_RET_ERR_SPECIFIC_BASE-1
ADL_FLH_RET_ERR_NO_ENOUGH_IDS	ADL_RET_ERR_SPECIFIC_BASE-2
ADL_FLH_RET_ERR_ID_OUT_OF_RANGE	ADL_RET_ERR_SPECIFIC_BASE-3

4.4 Specific GPRS Service Error Codes

Error Code	Error Value
ADL_GPRS_CID_NOT_DEFINED	-3
ADL_NO_GPRS_SERVICE	-4
ADL_CID_NOT_EXIST	5

4.5 Specific A&D Storage Service Error Codes

Error Code	Error Value
ADL_AD_RET_ERR_NOT_AVAILABLE	ADL_RET_ERR_SPECIFIC_BASE
ADL_AD_RET_ERR_OVERFLOW	ADL_RET_ERR_SPECIFIC_BASE - 1
ADL_AD_RET_ERROR	ADL_RET_ERR_SPECIFIC_BASE - 2
ADL_AD_RET_ERR_NEED_RECOMPACT	ADL_RET_ERR_SPECIFIC_BASE - 3

5 Resources

Here are listed the available resources of the Open AT® OS.

Resource name	Value
Maximum tasks count	30
Maximum running timers count per task	32
Semaphore resources	7

Make it wireless

WAVECOM S.A. - 3 esplanade du Foncet - 92442 Issy-les-Moulineaux Cedex - France - Tel: +33(0)1 46 29 08 00 - Fax: +33(0)1 46 29 08 08
Wavecom, Inc. - 4810 Eastgate Mall - Second Floor - San Diego, CA 92121 - USA - Tel: +1 858 362 0101 - Fax: +1 858 558 5485
WAVECOM Asia Pacific Ltd. - Unit 201-207, 2nd Floor, Bio-Informatics Centre – No.2 Science Park West Avenue - Hong Kong Science Park, Shatin - New Territories, Hong Kong

www.wavecom.com